Oscar Contreras

Seth Thomas Giovannetti

Brennan Reid Steuwe

**Server Based Chat - Team Report**

Compilation and Execution Instructions:

See README.md for details.

Protocol Messages and Semantics:

* HELLO (Client-ID-A): Initiates the process for Client A to be authenticated and registered with the server.
* DECLINED: Sent by the server if a client is not a subscriber to the chat (i.e. their ID is not found in SubscriberList.txt).
* CHALLENGE (Rand): Sent by the server to challenge the client to authenticate itself. rand is a random number generated by the server. A new rand is generated at every challenge.
* RESPONSE (Client-ID, Response): Response to the challenge, sent by the client to authenticate itself.
* AUTH\_SUCCESS (Rand-Cookie, Port-Number): Sent by the server to notify the client authentication is successful. rand\_cookie is a random number generated by the server, and the port number is a TCP port assigned by the server for subsequent connection by the client.
* AUTH\_FAIL: Sent by the server to notify the client that authentication has failed.
* CONNECT (Token): Sent by the client to the server. The token is the cookie previously sent by the server.
* CONNECTED: Sent by the server to notify the client it has been connected.
* DISCONNECT: Sent by client to disconnect from the chat server.
* CHAT\_REQUEST (Client-ID-B): Sent by client A to the server to request a chat session with Client B.
* CHAT\_STARTED (Session-ID, Client-ID-B): Sent by the server to notify client A that a chat session with client B has started. Session-ID is an identifier assigned to the unique pair of clients involved in the chat.
* CHAT (Message): Exchanged between the clients, relayed by the server. Carries the actual chat message.
* UNREACHABLE(Client-ID-B): Sent by the server to client A to notify that the client B is not available for chat.
* END\_REQUEST: Sent by any of the clients involved in the chat session to request a termination of their current session.
* END\_NOTIF: Sent by the server to notify both clients involved in a session that the session has been terminated.
* HISTORY\_REQ (Client-ID-B): Sent by client A to request the history of past chat messages with client B.
* HISTORY\_RESP (Session-Number, Sending Client-ID, chat-message): Sent by the server to the client who requested the history. Session-Number indicates an instance of the chat session. Sending Client-ID is the ID of the client who sent the chat message, and chat message is the chat message in the history. There is one HISTORY\_RESP message for each chat message in the history.

Each protocol message is assigned a unique bytecode, and this bytecode along with any required parameters are sent into the UDP or TCP socket for a specified connection (more on this in the next section).

Syntax and Formatting Decisions:

Rather than implementing the protocol messages exactly as outlined in the requirements we designed our own format for sending data, and as well as a command syntax bolstered by python’s prompt\_toolkit module. Two routines which characterize our message format are the net.sendUDP and net.sendTCP methods in net.py. A sender wishing to send a message has to convert the protocol bytecode and any other parameters into a sequence of bytes and call one of these methods to send the data into an appropriate socket. Our format or syntax for messaging requires the sending socket as the first argument for sendUDP and sendTCP, followed by the generated message bytes. The sendUDP routine must also include an address to send to, as there is no established connection in this transport-layer protocol as there is in TCP. The bytes are generated by a routine in byteutil.py which takes the bytecode as the first parameter, followed by whatever arguments are used in the protocol message. This formatting approach is advantageous due to the ease of extending the program and adding new protocol messages. The advantage of our command syntax is that it allows us to encapsulate the interactive protocols and allowed us to create a more natural and useful prompt. For example, to initiate a session, a client must type ‘chat clientID’ and the program will invoke sendTCP appropriately as we have described.

Overview of Source Code:

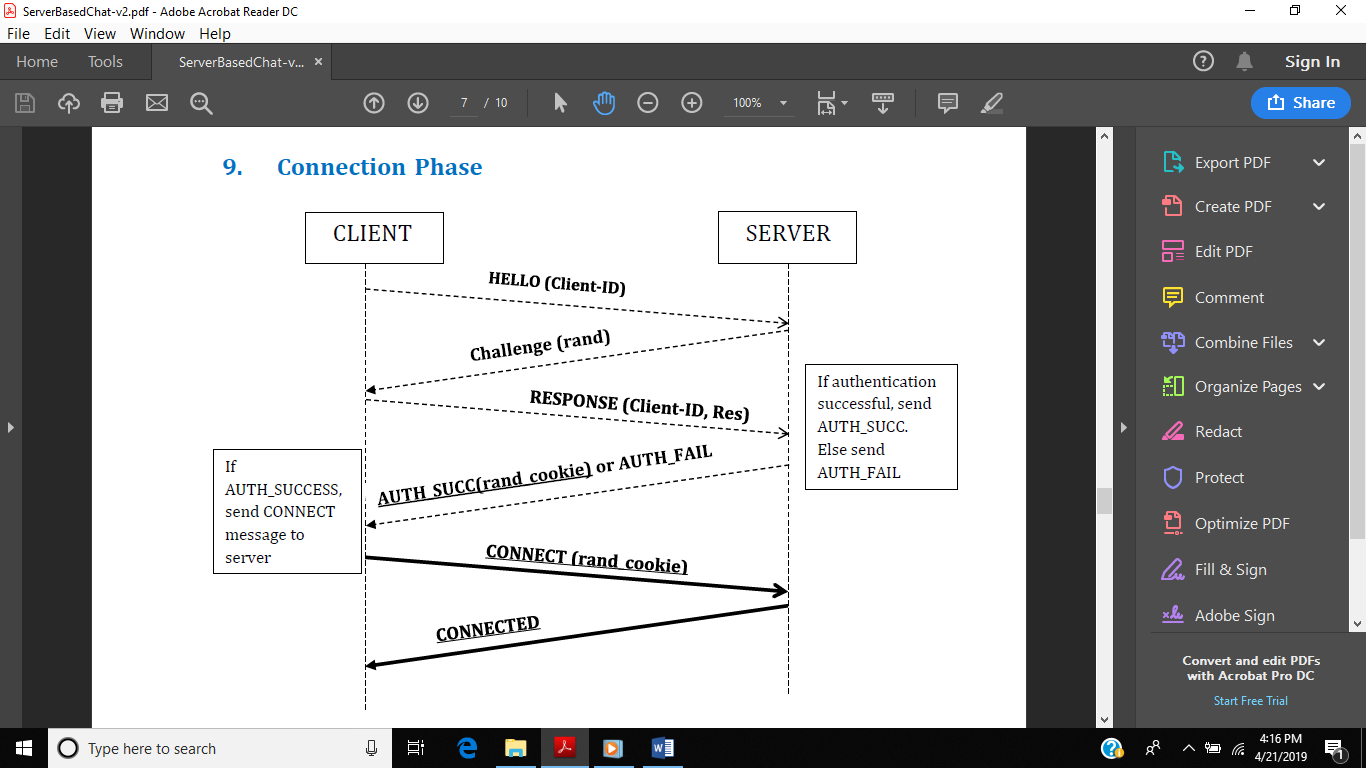
* byteutil.py: Contains various methods for coercing objects to and from bytes to be sent over the network transport protocols.
  + Messages are delimited by byte b`\x03`
  + Internal fields (client-ID and actual message) are delimited by byte b`\x00`
* Client.py: Base code for client.
  + Clients maintain their own ID, current session and the ID of the client they are chatting with (these attributes are None if not in a chat)
  + Fundamental commands for interactive prompt
    - ‘chat Client-ID-B’ to initiate chat, ‘end chat’ to exit chat
    - ‘history Client-ID-B’ to request history
    - ‘exit’ or ‘disconnect’
* Codes.py: Defines codes used to identify protocol messages and other reserved bytes.
* start\_server.py: Execute to start up the chat server.
* start\_client.py: Execute to create an interactive client.
* Server.py: Base code for server.
  + Maintains dictionaries of addresses-to-clients and clients-to-connections (also has dictionaries of clients-to-tokens or clients-to-challengeRands for authentication)
  + Uses hash of client-IDs to generate a unique session-ID. This identifies a history file to write chat messages to.
  + Useful server commands
    - ‘vars’ - show server variables
    - ‘net’ - show network status and open TCP connections
    - ‘codes’ - show reserved bytecodes
* crypto.py: Code necessary for client authentication functionality. Contains implementation of A3 algorithm (SHA256 hash) and routines for client key retrieval.
* history.py: Code necessary for retrieving past chat history or storing new messages into chat history.
* listener.py: Code to handle the receiving of UDP and TCP messages
  + Implements a “spin” mechanism to handle segments which may exceed our defined maximum packet size of 4KB.
  + UDP listener is implemented as a class to allow threading on the server side. On the client only a simple UDP socket is needed for use in the connection phase.
  + Routine tcpListen is run in a thread for each TCP connection to listen on a socket.
* net.py: Implements base code for sending and receiving UDP and TCP messages as well as opening UDP and TCP sockets.
* Other files provide backbone utilities for the program or are unused.

Authentication:

Although it was not required for our revised instructions, we implemented the challenge and response authentication scheme laid out in the initial project. The following sequence of steps summarizes the process needed to establish a TCP connection.

1. When a client program begins and a client\_id is entered, the client sends a HELLO message over UDP to the server.
2. On receiving the HELLO from the client, the server generates a random number and sends a CHALLENGE message over UDP if the client is a subscriber. If the client is not a subscriber, the server sends a DECLINED message over UDP.
3. On receiving the CHALLENGE from the server, the client does the following:
   1. If needed, generates a secret ciphering key and stores this in a folder named ‘keys’ (the folder will contain key files with the notation client\_id.key).
   2. Runs an A3 authentication algorithm (SHA256) with the random number sent by the server and the client\_id as inputs.
   3. Sends a RESPONSE message over UDP to the server, containing the result RES of the A3 algorithm.
4. On receiving the RESPONSE from the client, the server:
   1. Runs the same A3 authentication algorithm using the random number from step 2 and the stored client key.
   2. Compares its result XRES with the result given by the client.
      1. If these are not equal, the server sends an AUTH\_FAIL message over UDP to the client and an error is raised.
      2. If there are equal, the server sends an AUTH\_SUCC message over UDP to the client with a token (rand\_cookie, another generated random number) and TCP port to establish a connection.
5. On receiving an AUTH\_SUCC message from the server, the client opens a TCP socket and sends to the server a CONNECT message over TCP containing the token.
6. On receiving the CONNECT message, the server verifies the token and sends a CONNECTED message to the client over TCP.

Sequence Diagram of Connection Establishment:



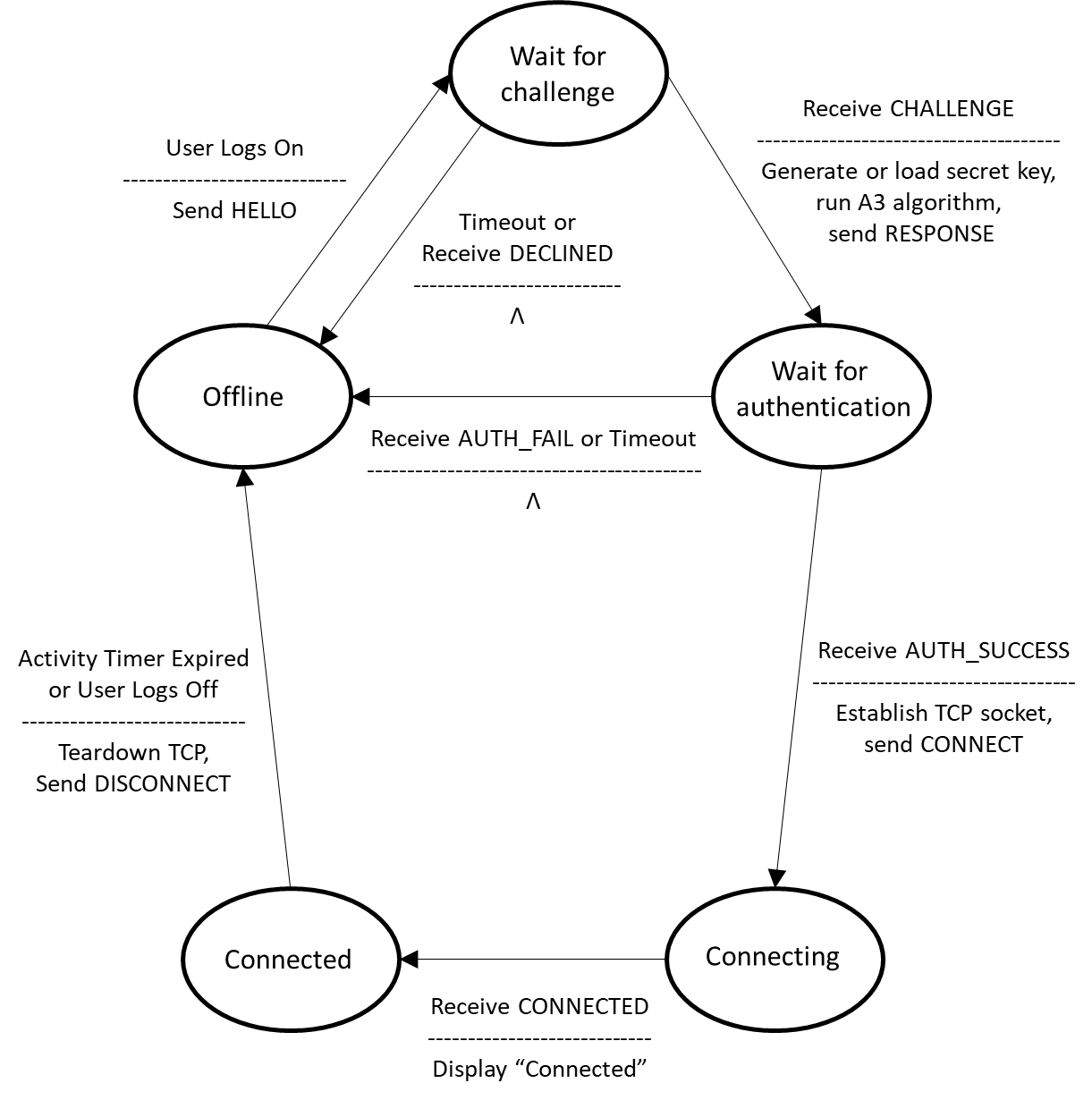
Hardware Setup and Configuration:

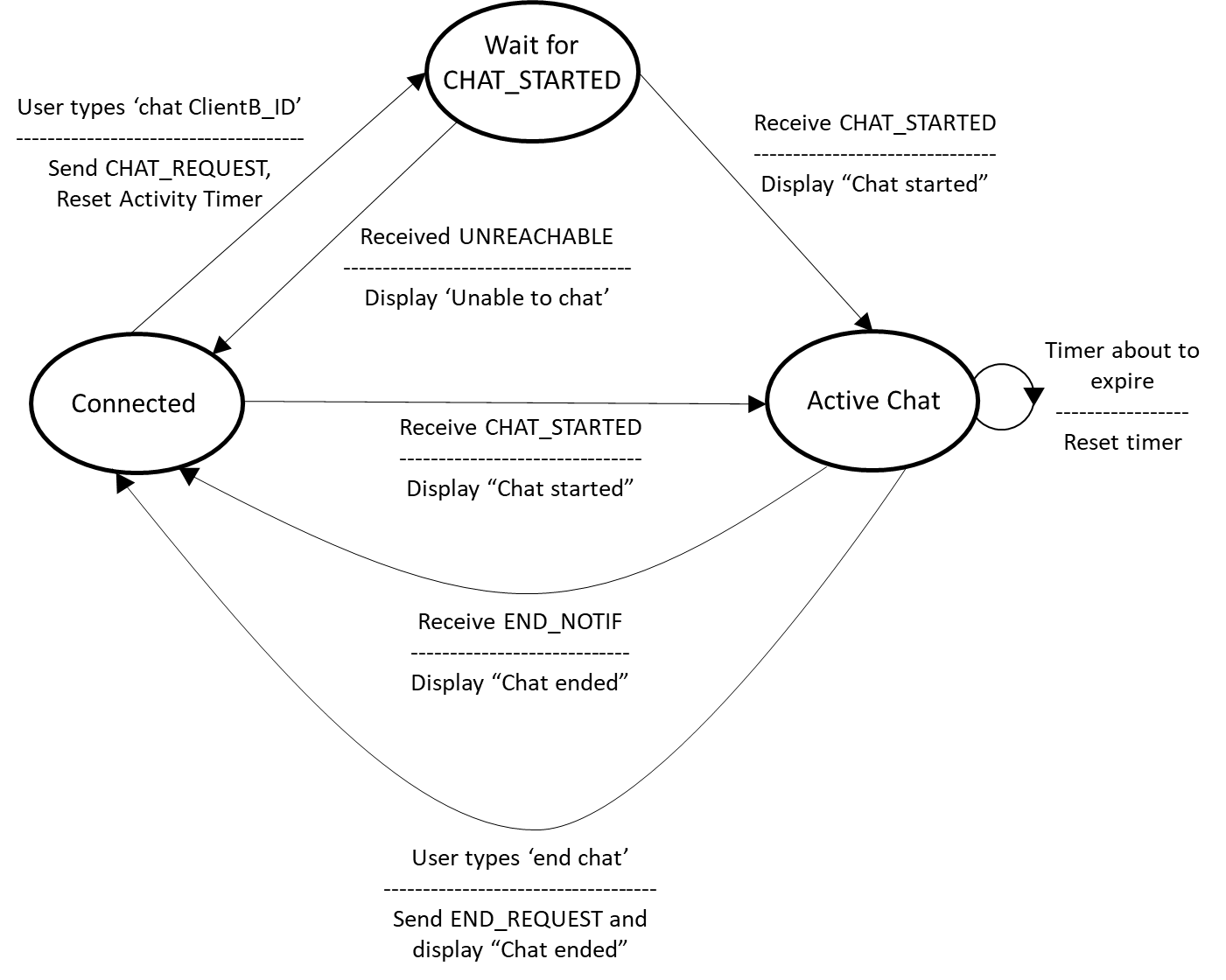
OS: Windows 10

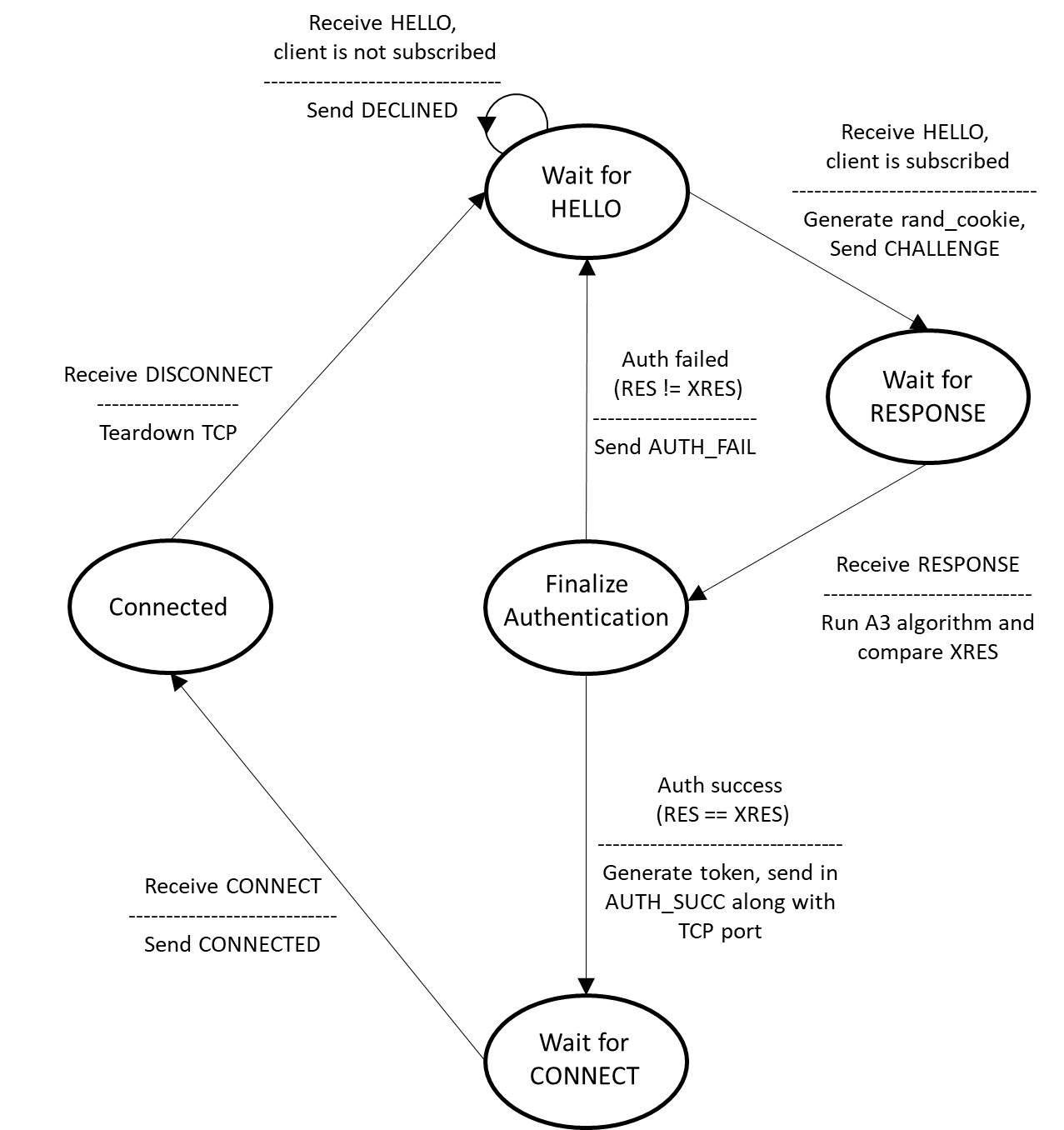
Python Version: 3.7

Required Modules: prompt\_toolkit

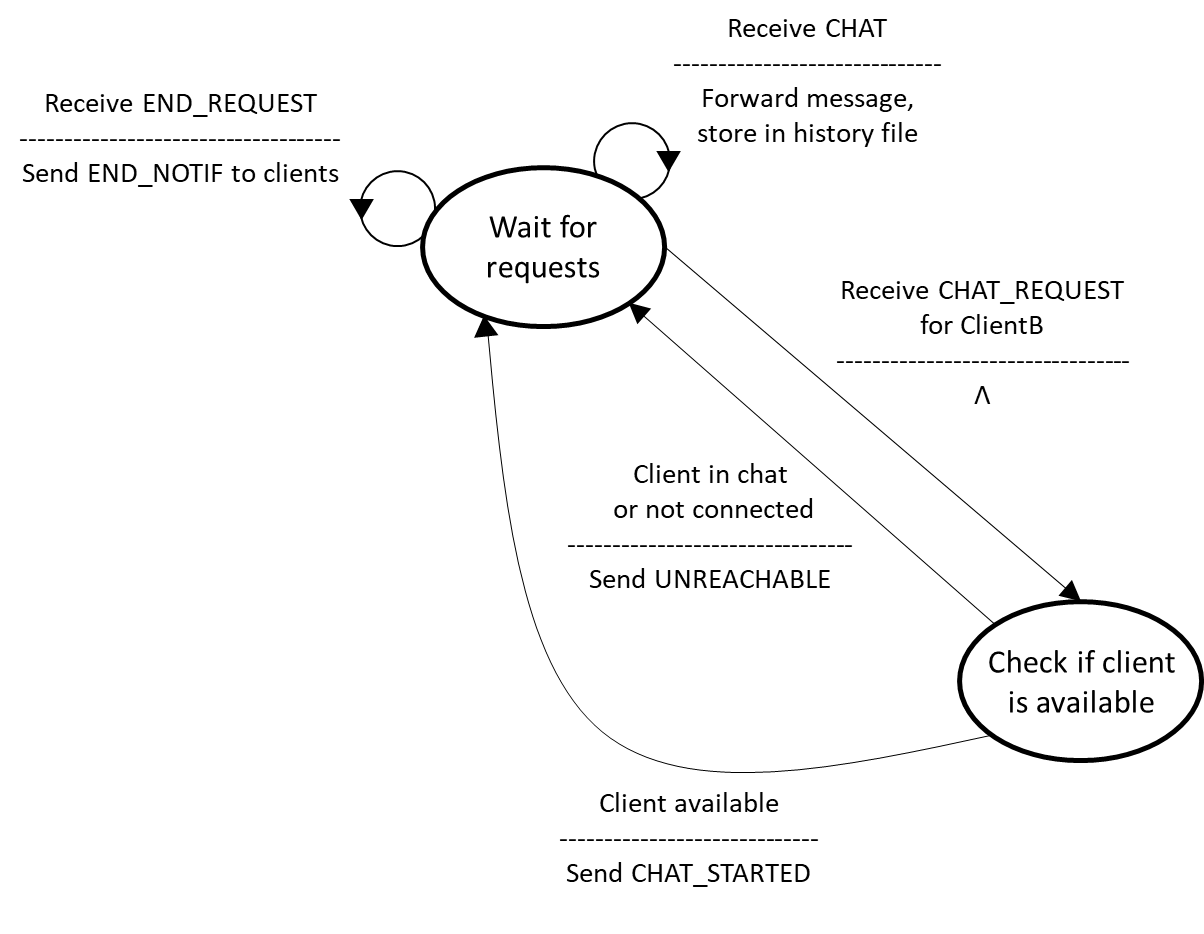
Design Documents:

**Client - Connection Phase:**

**Client - Chat Phase:**

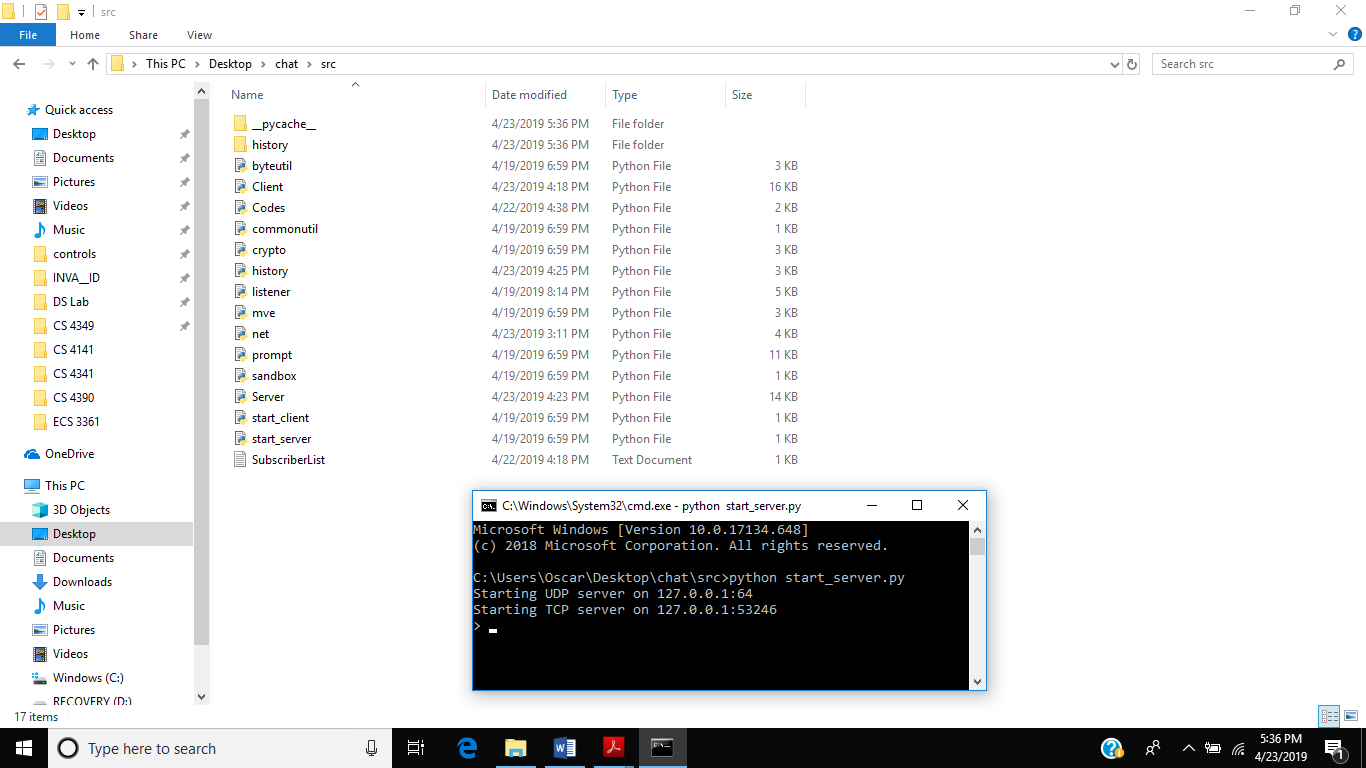
**Server - Connection Phase:**

**Server - Chat Phase:**

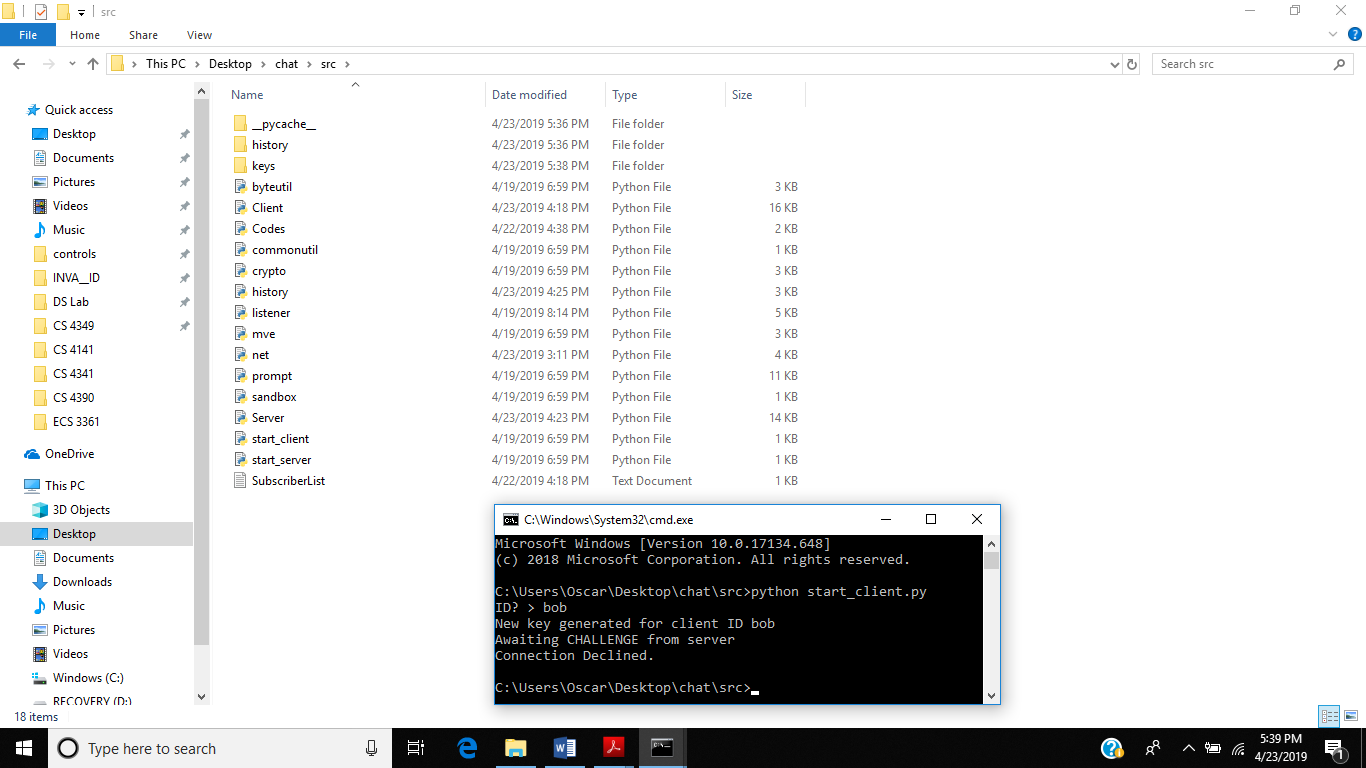


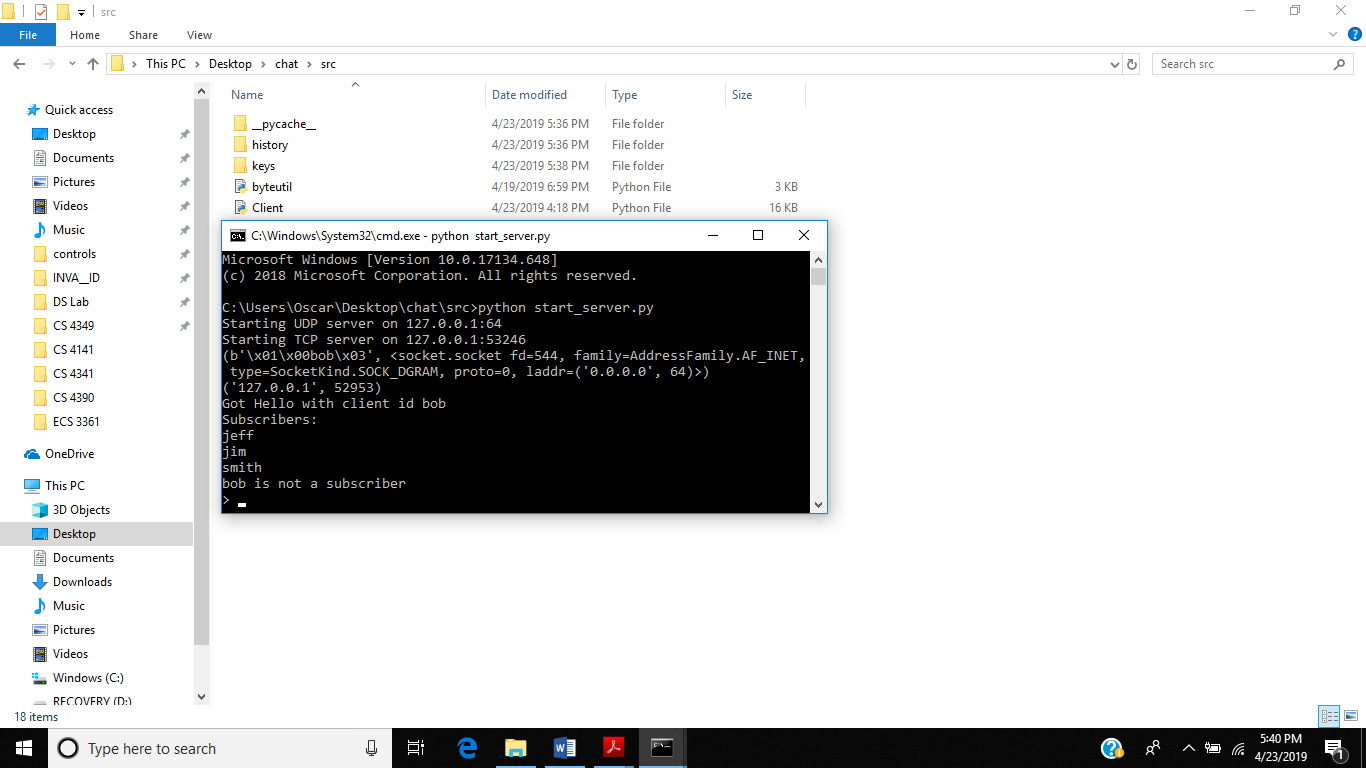
Screenshots and Validation Scenarios:

**Scenario 0: Running Server**

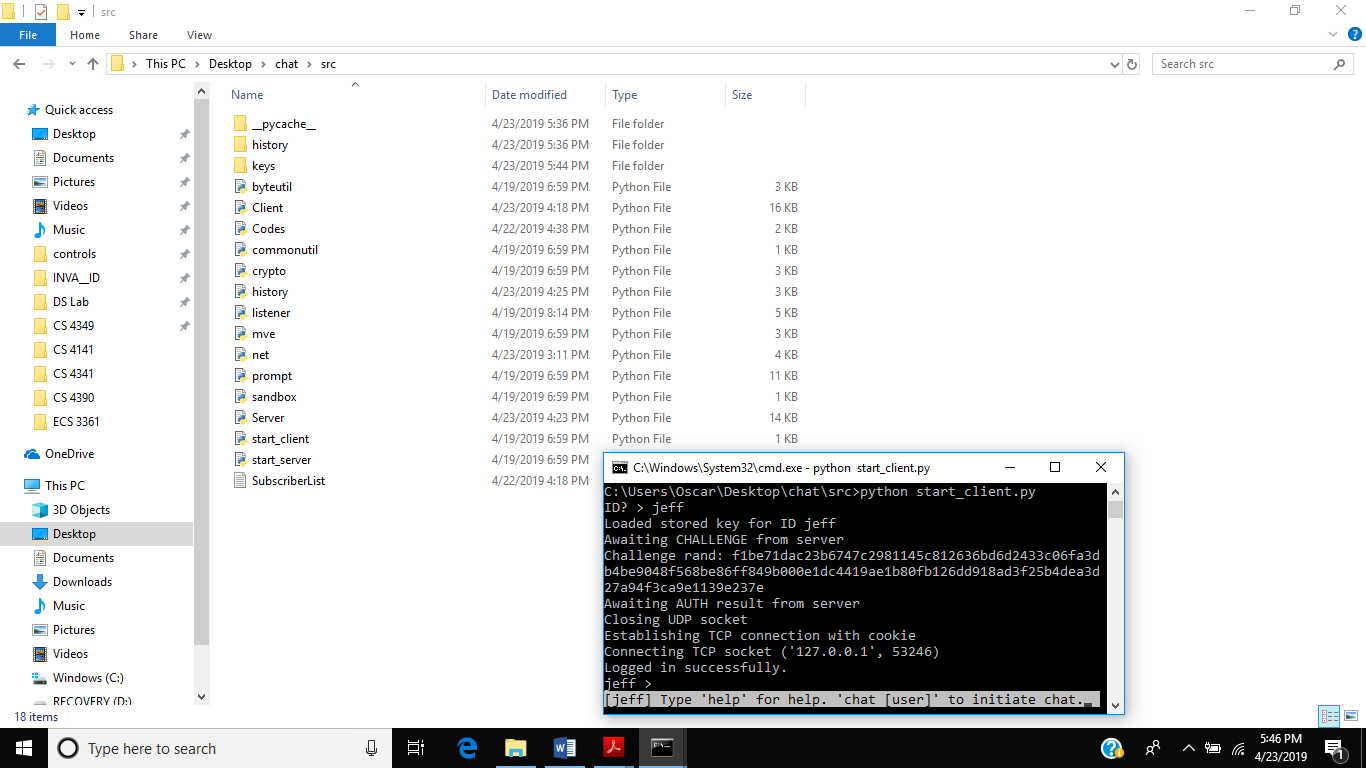
Server:

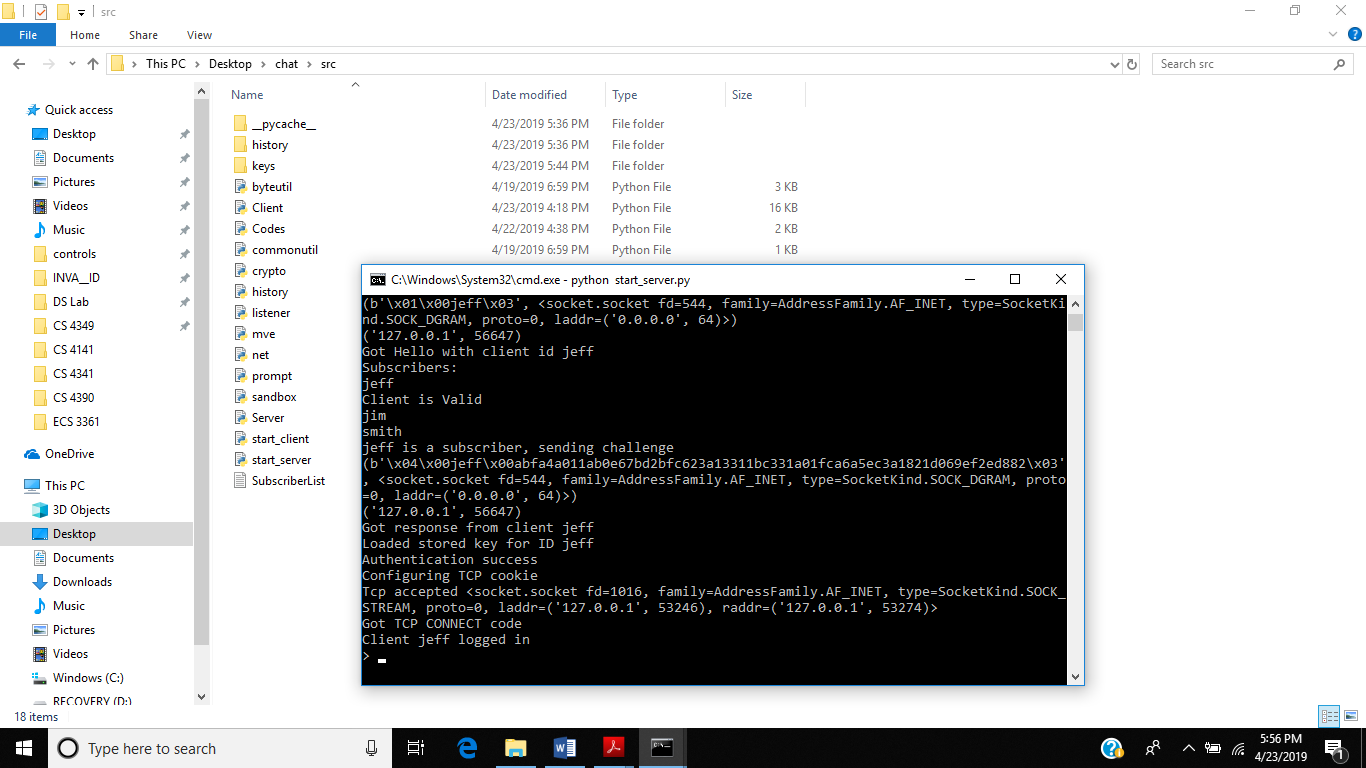
**Scenario 1a: Attempted connection by non-subscribing Client**

Client:

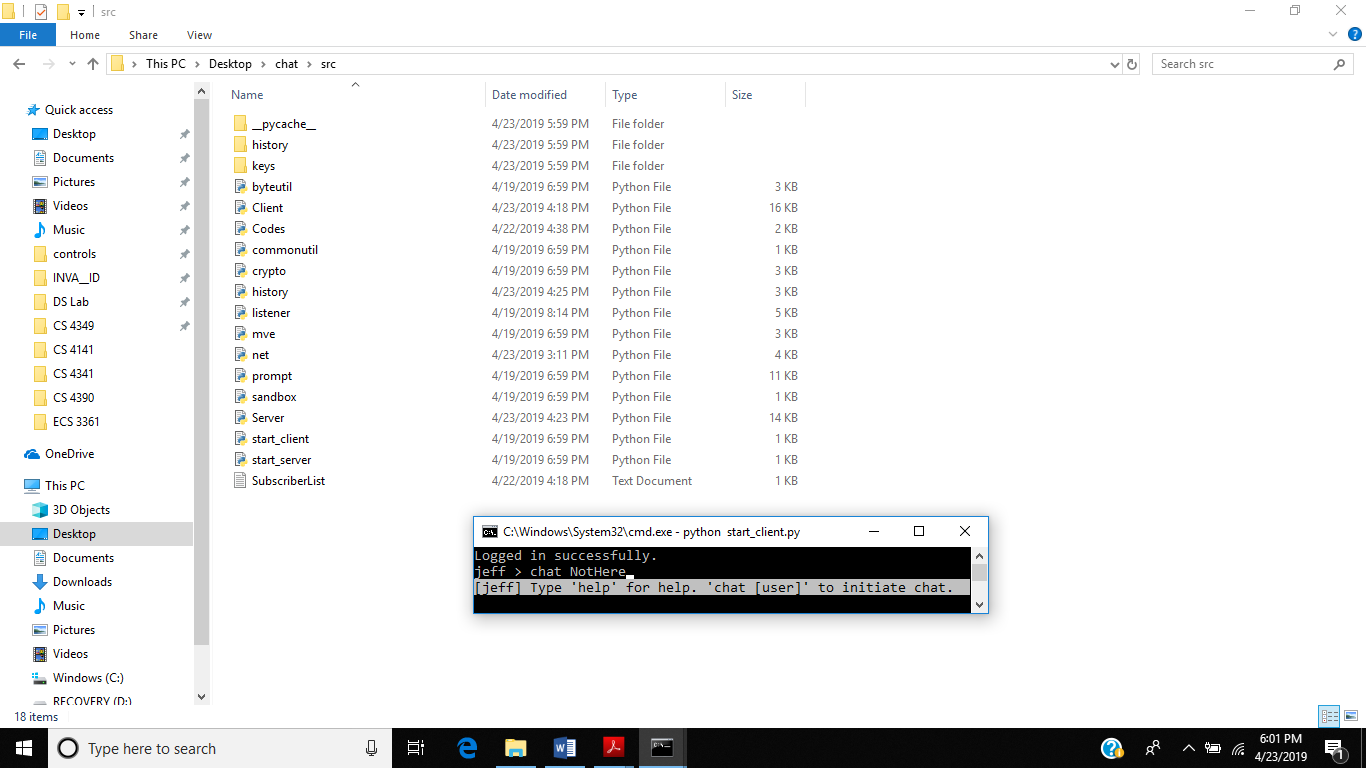
Server:

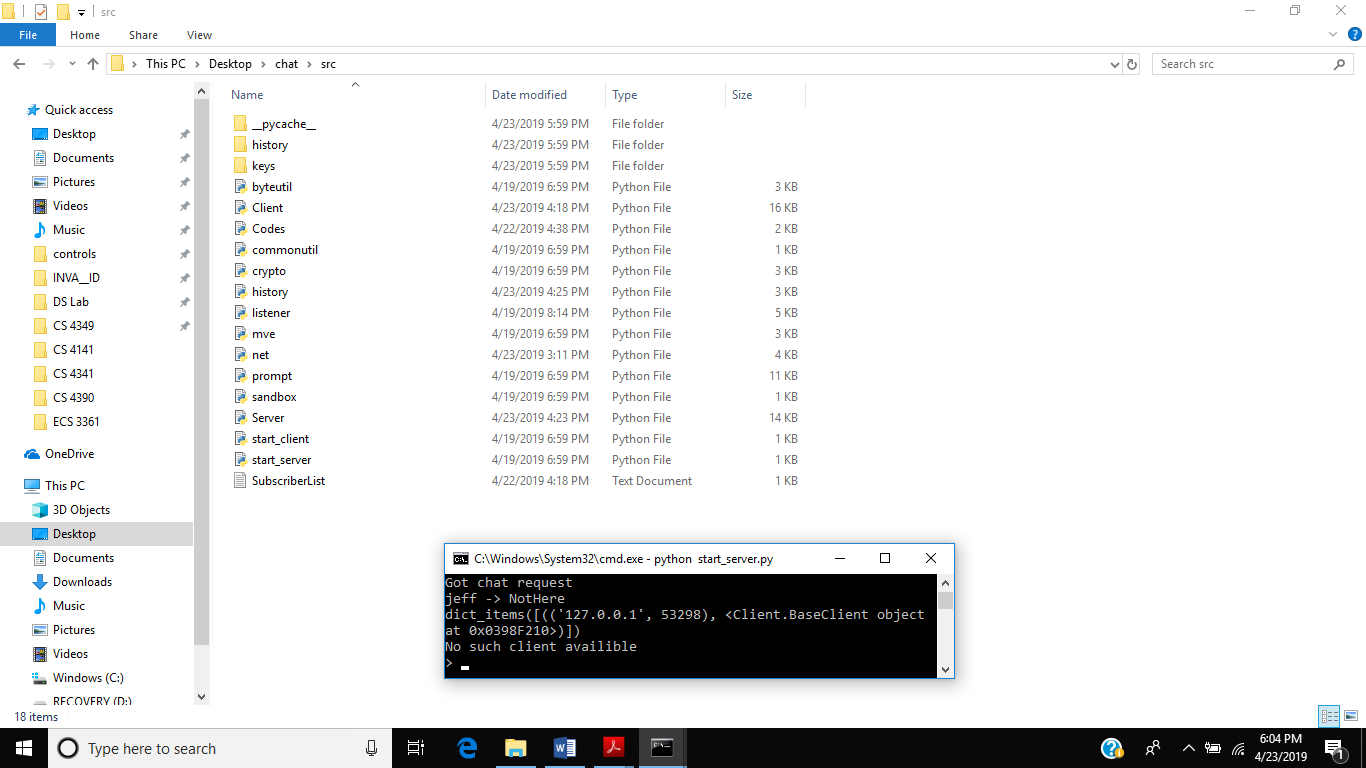
**Scenario 1b: Attempted connection by subscribing Client**

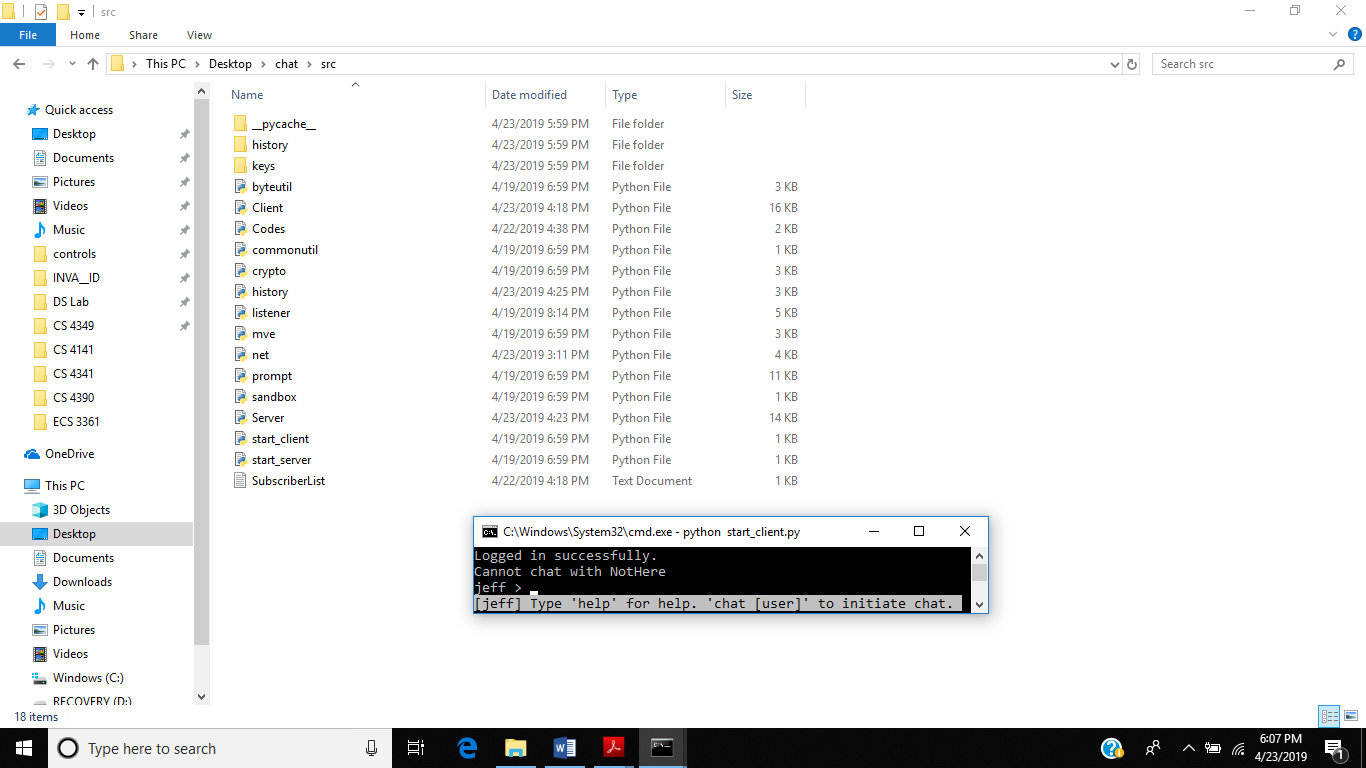
Client:

Server:

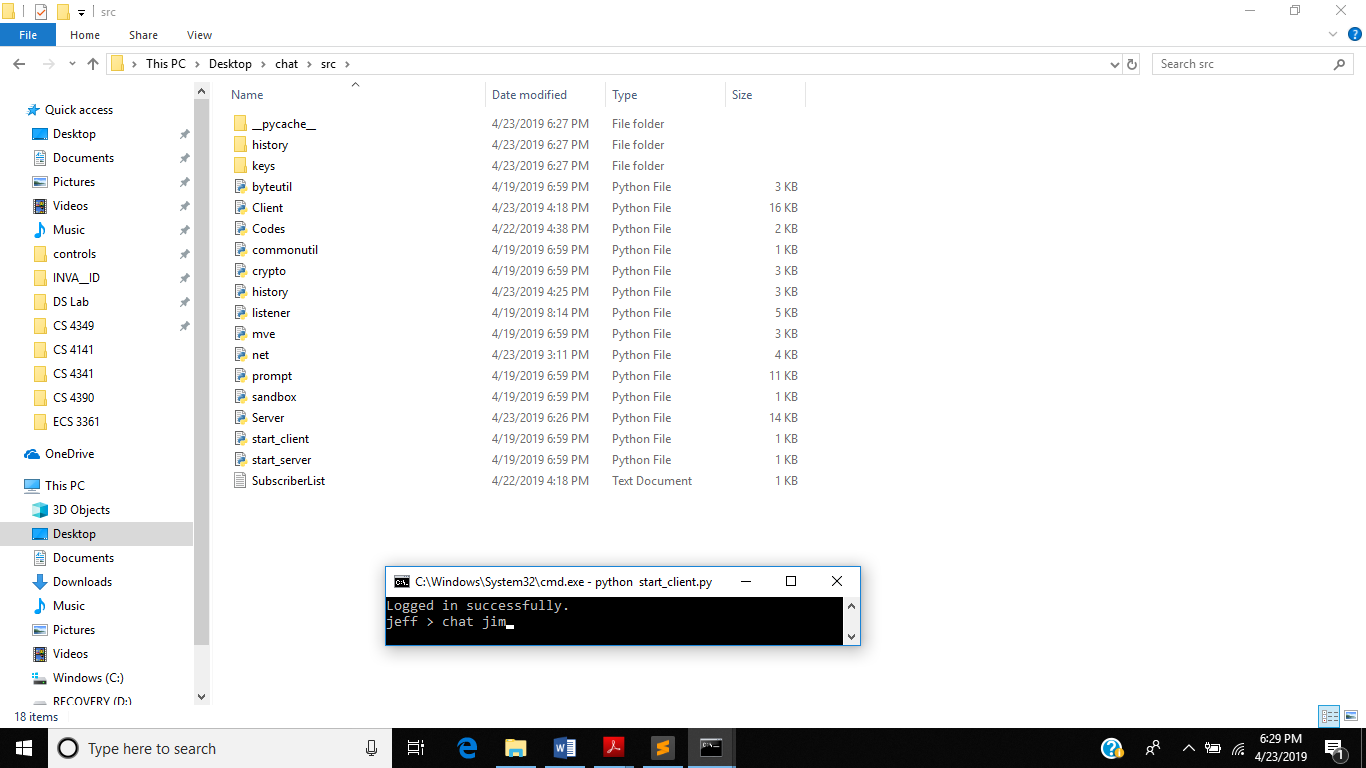
**Scenario 2a: Connected Client A attempts to chat with unreachable Client B**

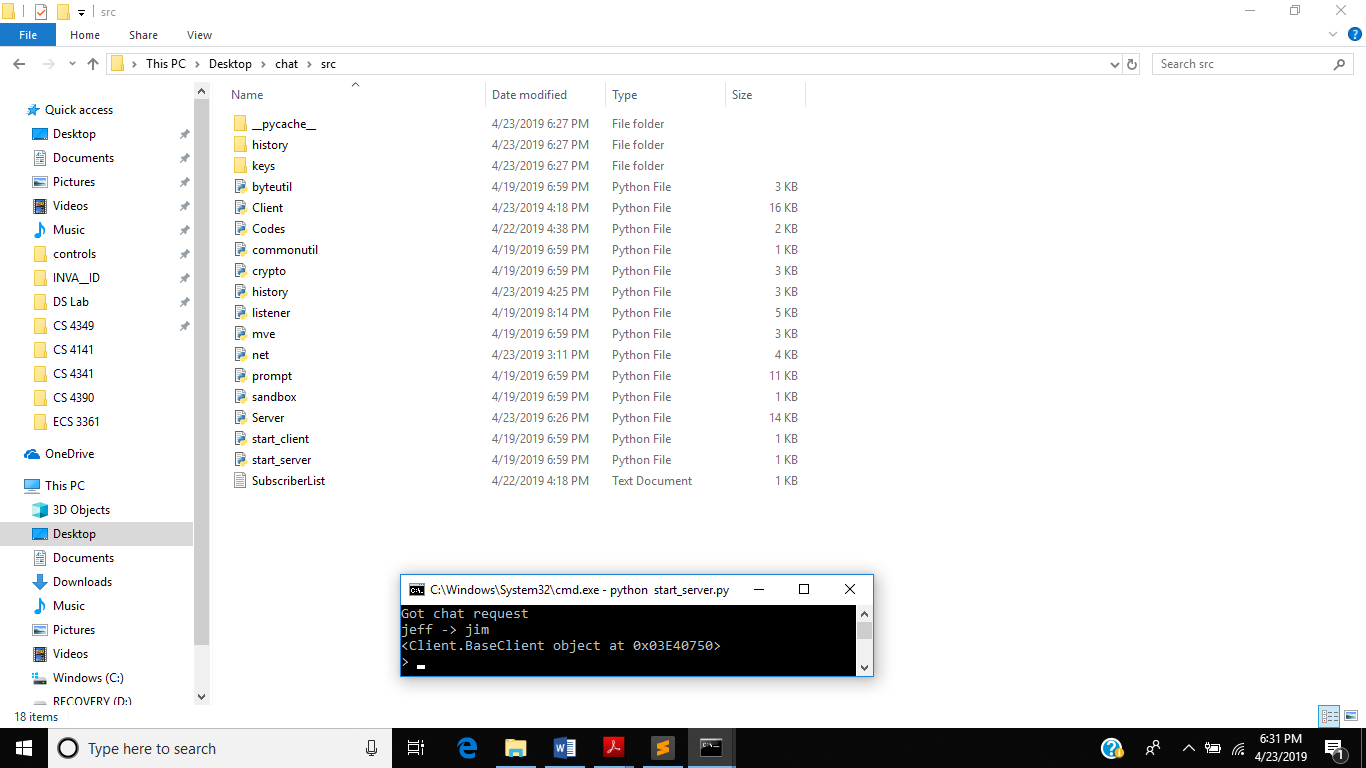
 Client Input:

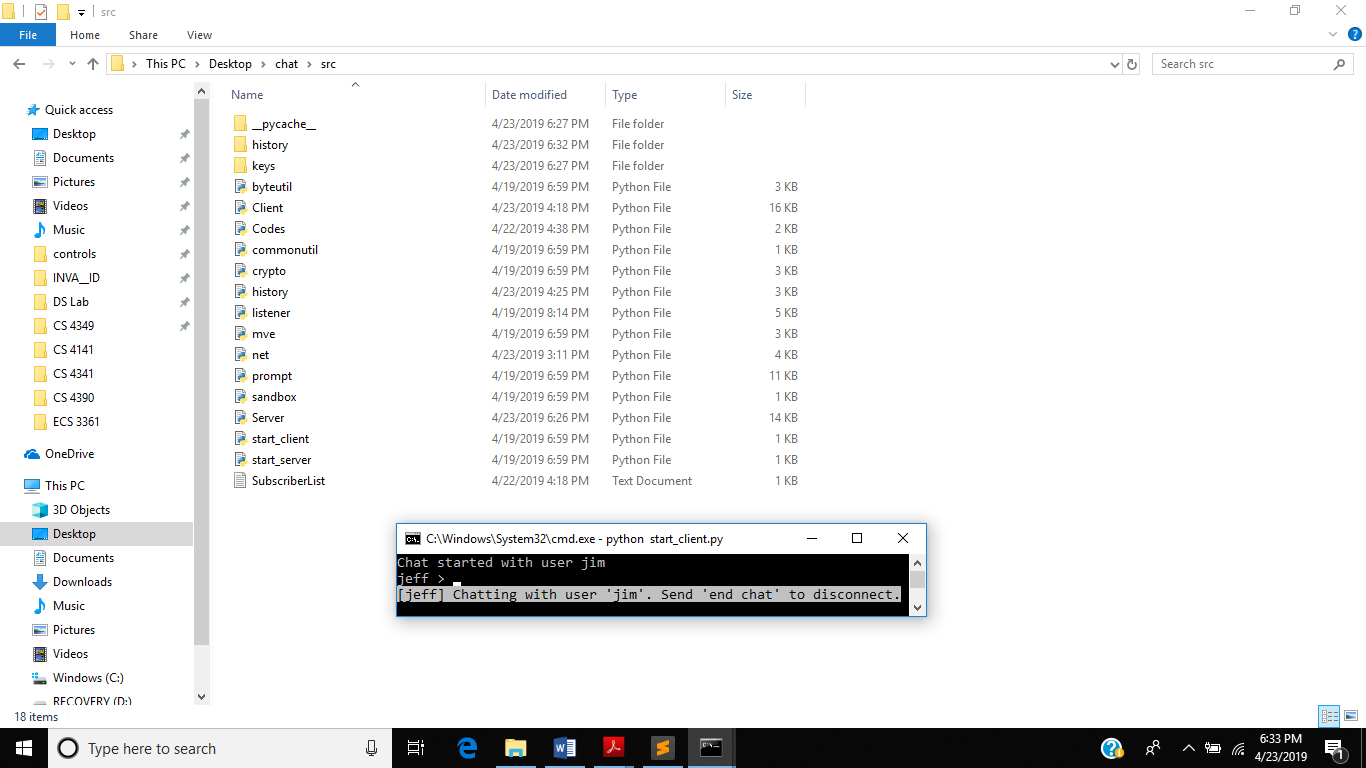
Server:

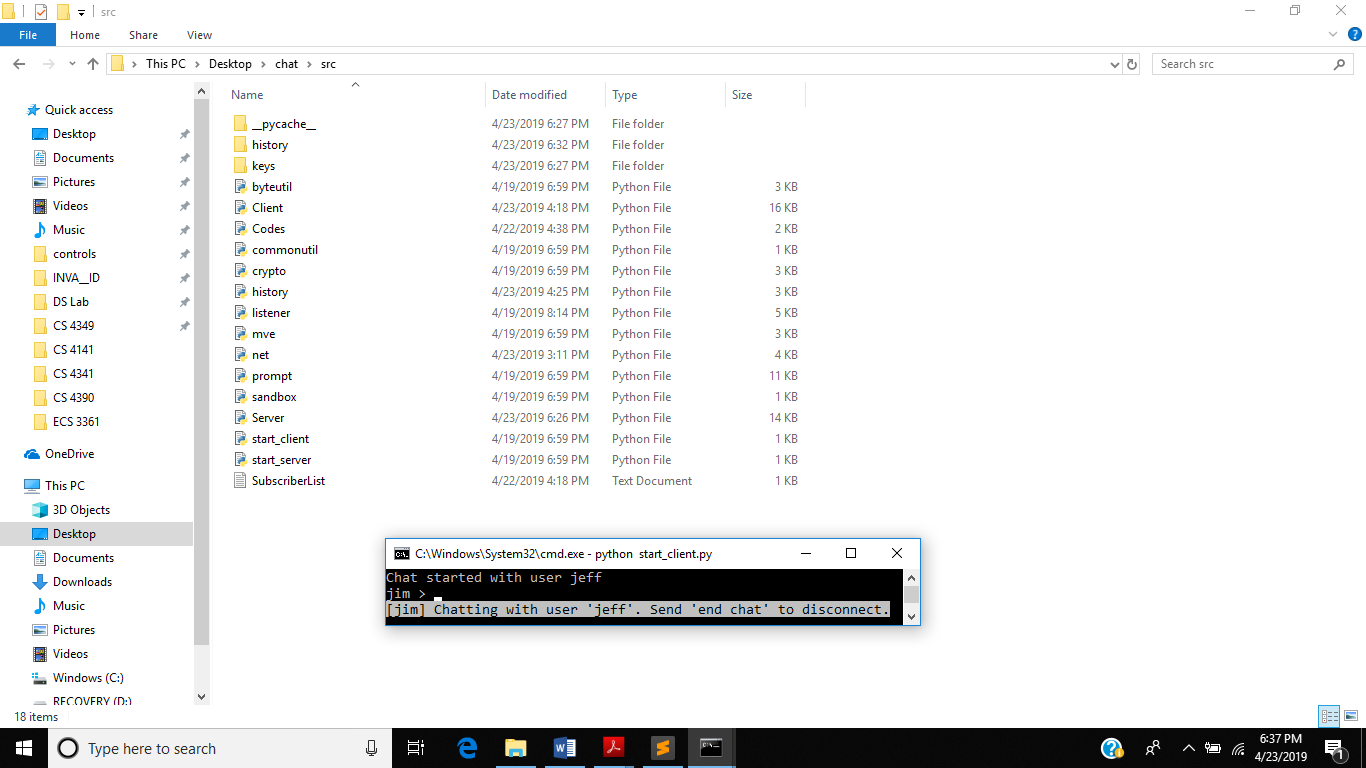
 Client Output:

**Scenario 2b: Connected Client A attempts to chat with connected and available Client B**

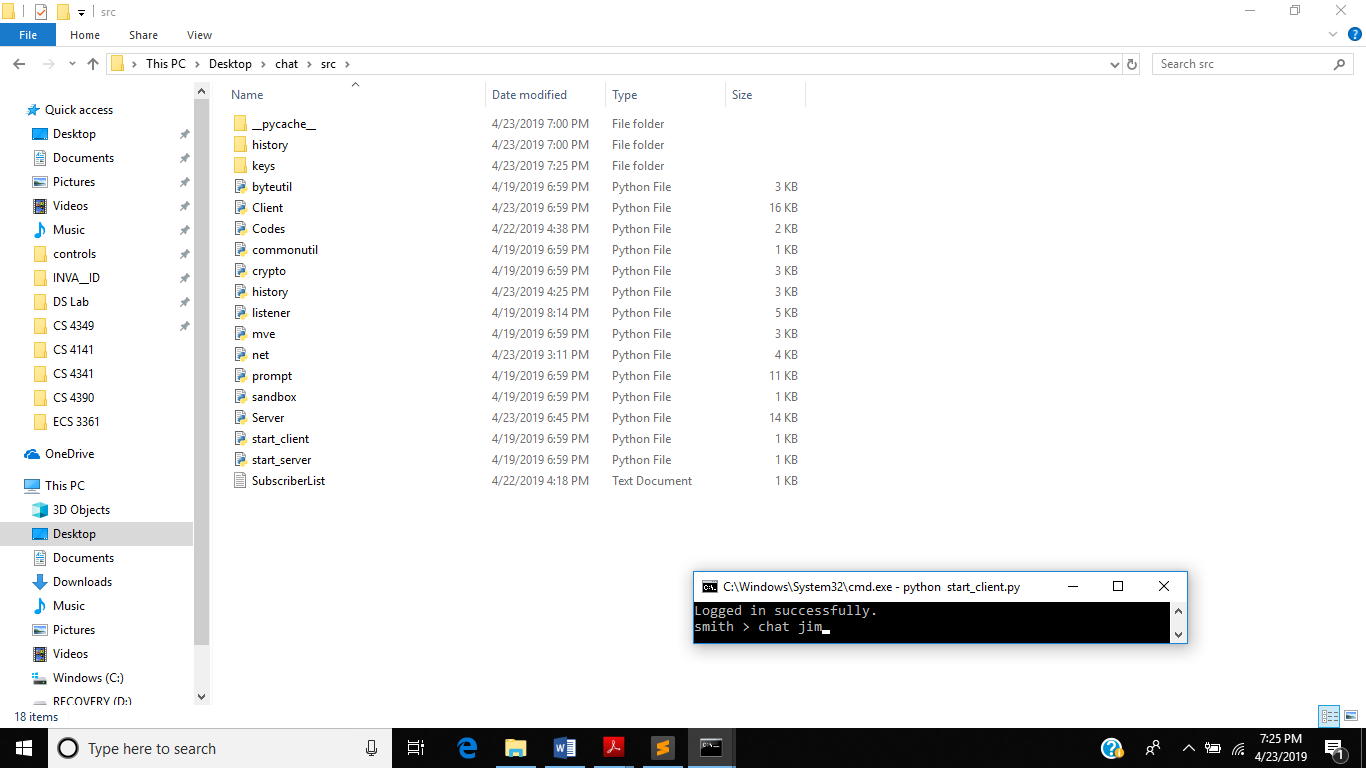
 Client-A Input:

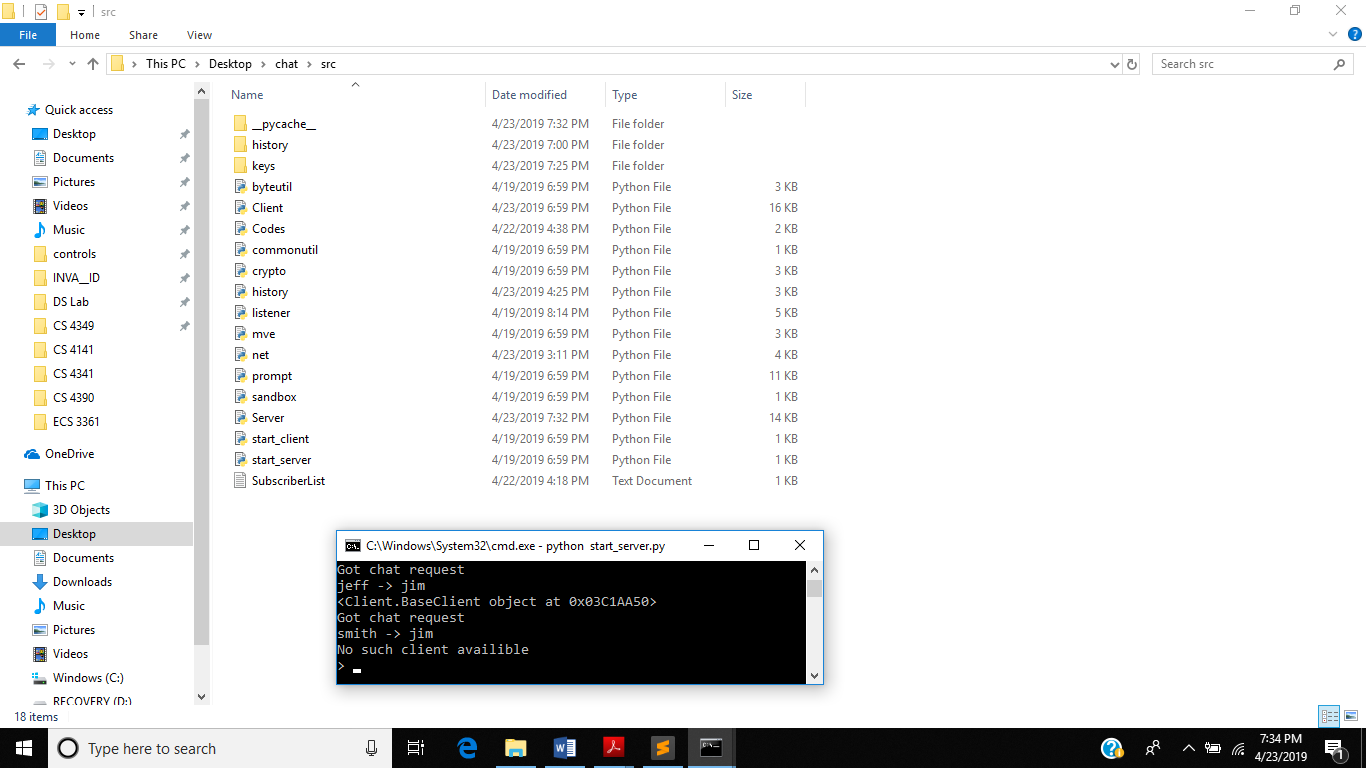
 Server:

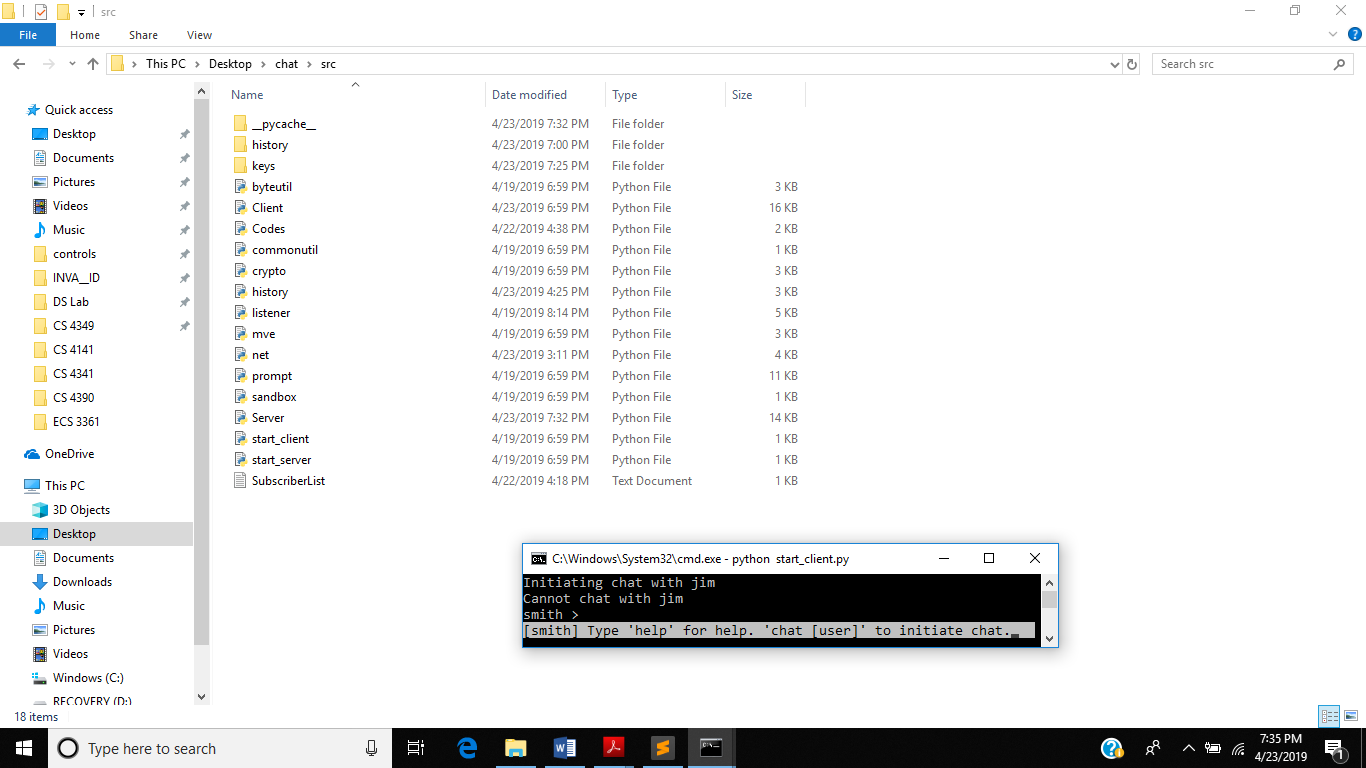
 Client-A Output:

 Client-B Output:

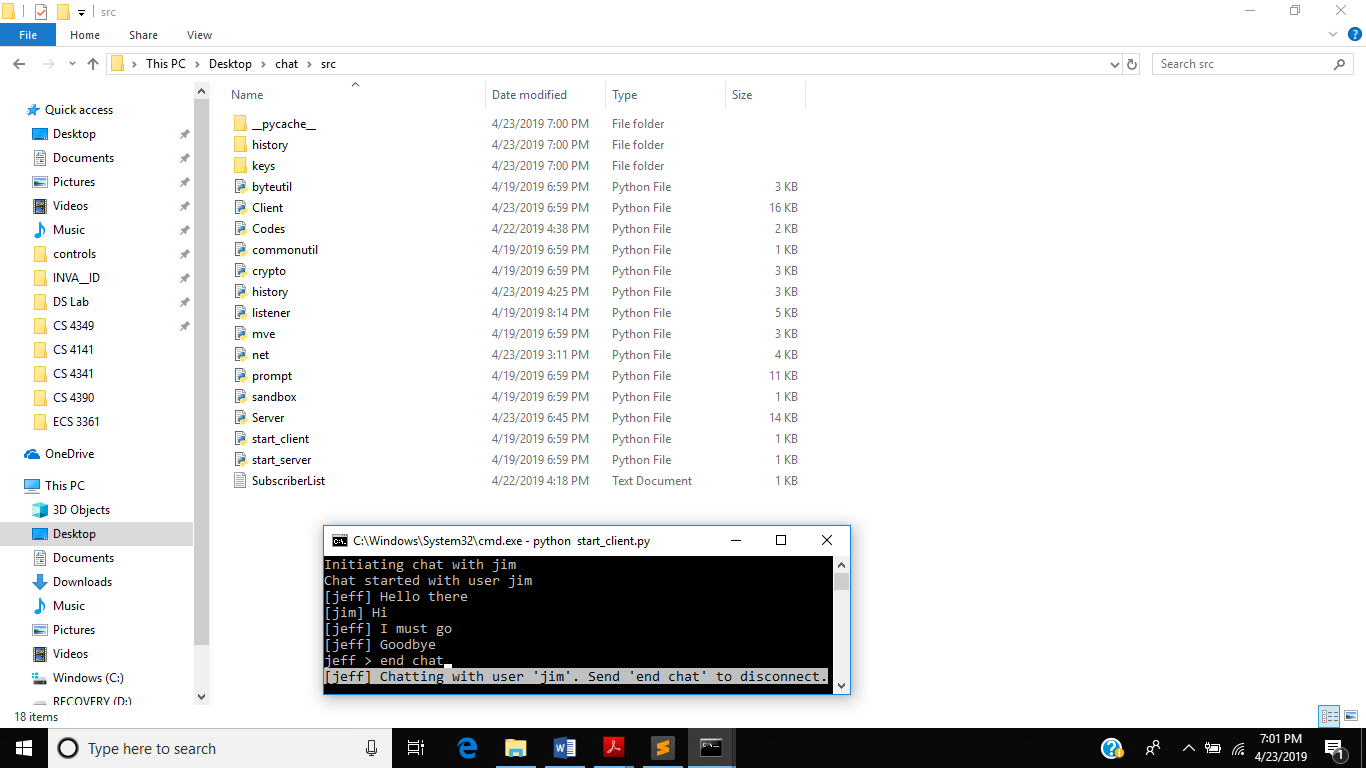
**Scenario 2c: Client C attempts to chat with an already chatting Client**

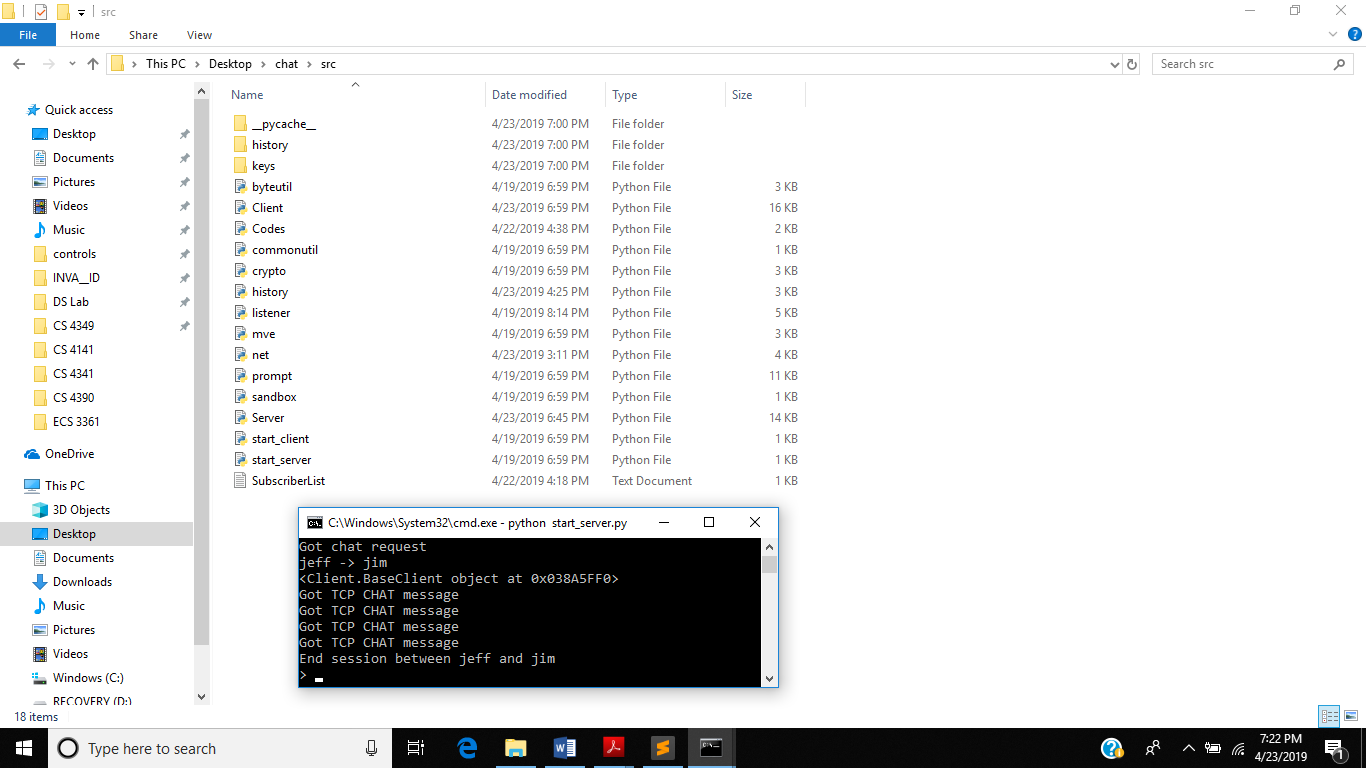
 Client-C Input:

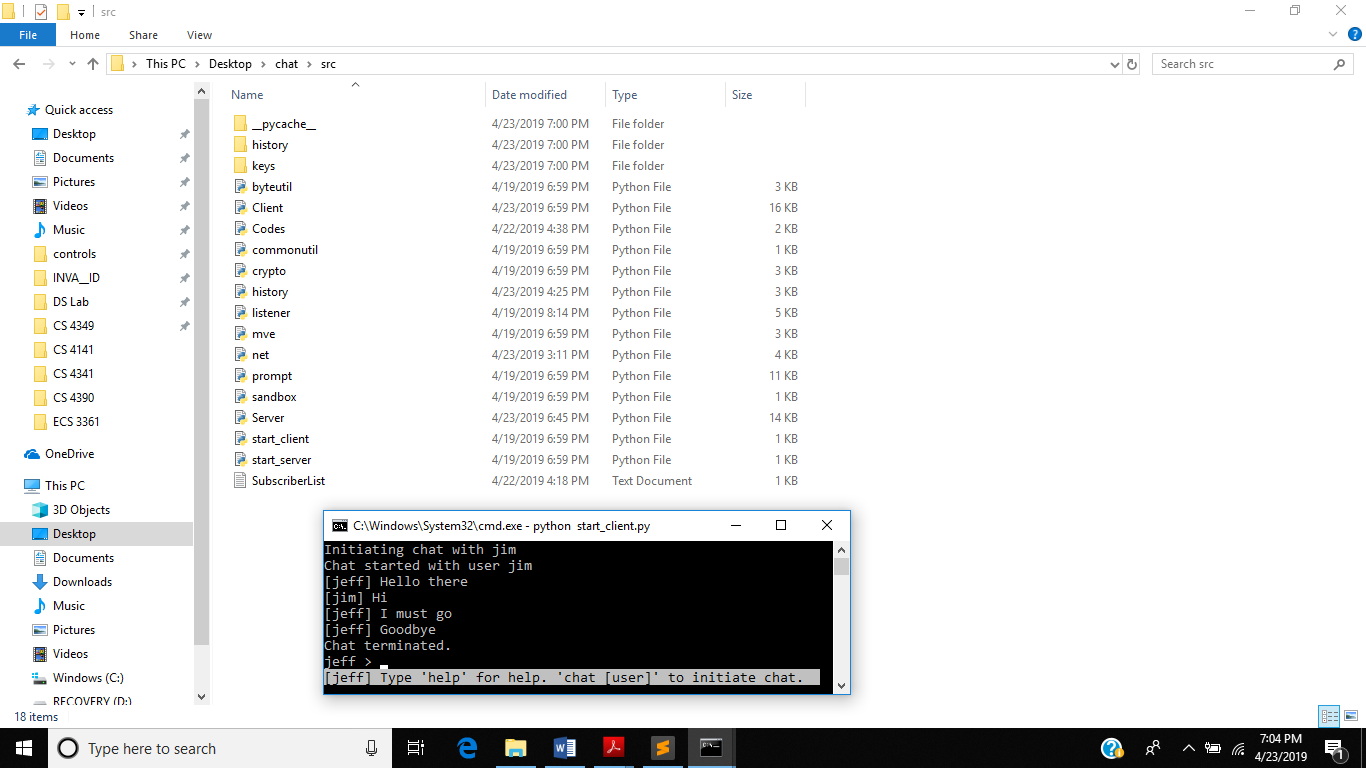
 Server:

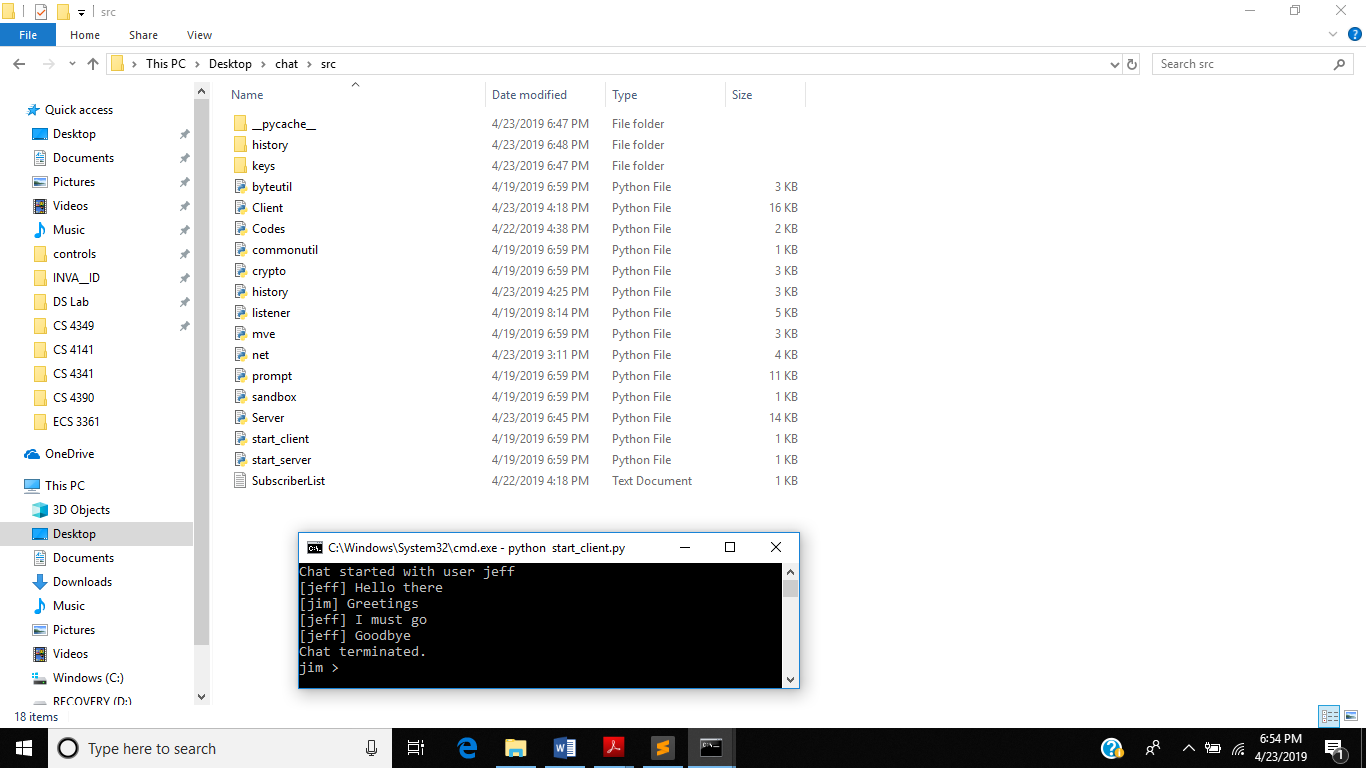
 Client-C Output:

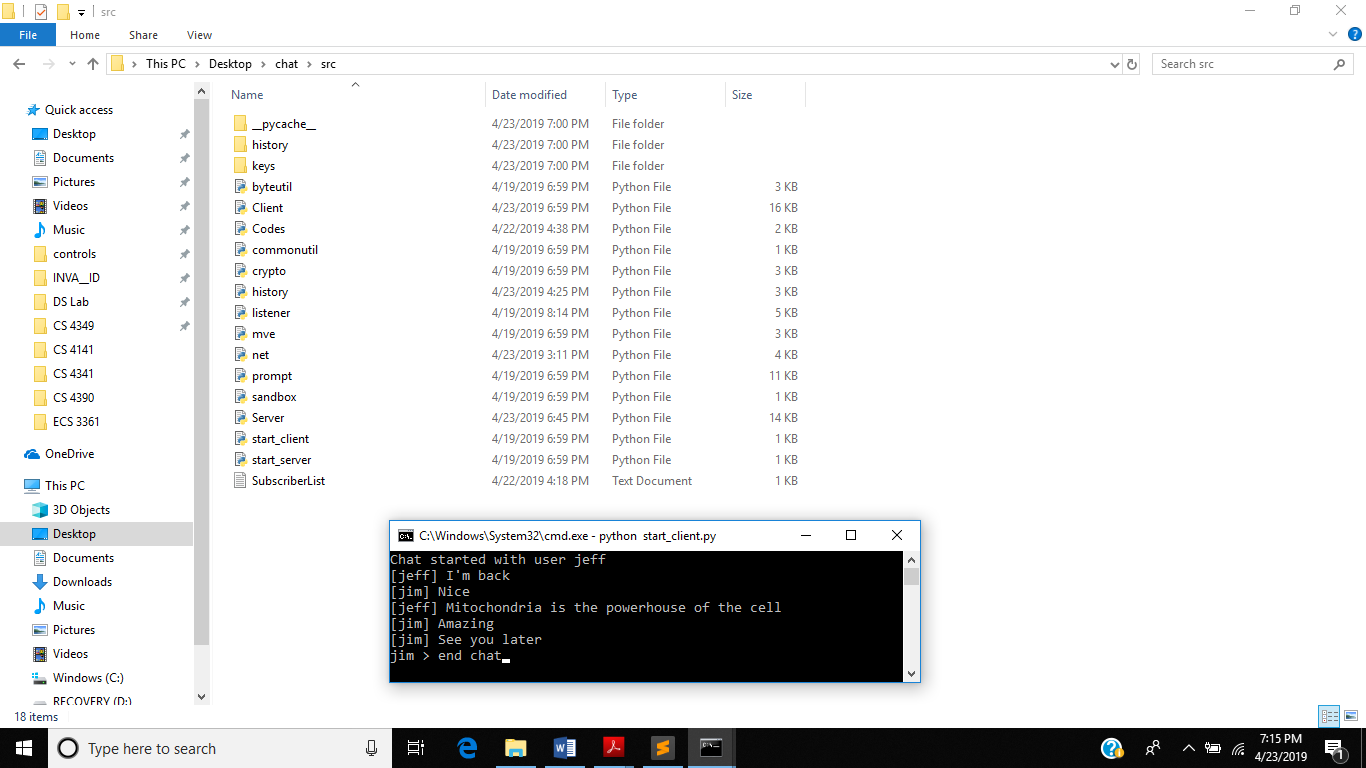
**Scenario 3a: Chat ended by initiating Client A**

 Client-A Input:

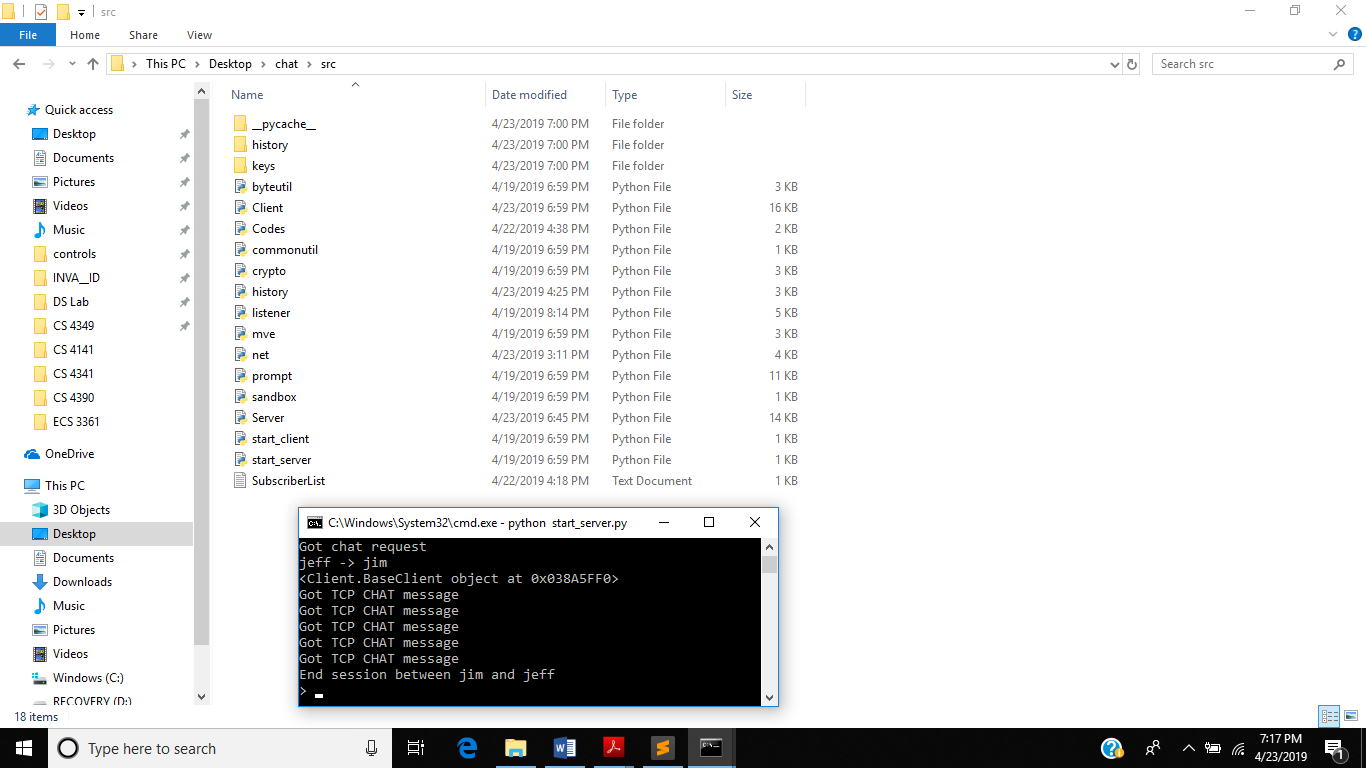
 Server:

 Client-A Output:

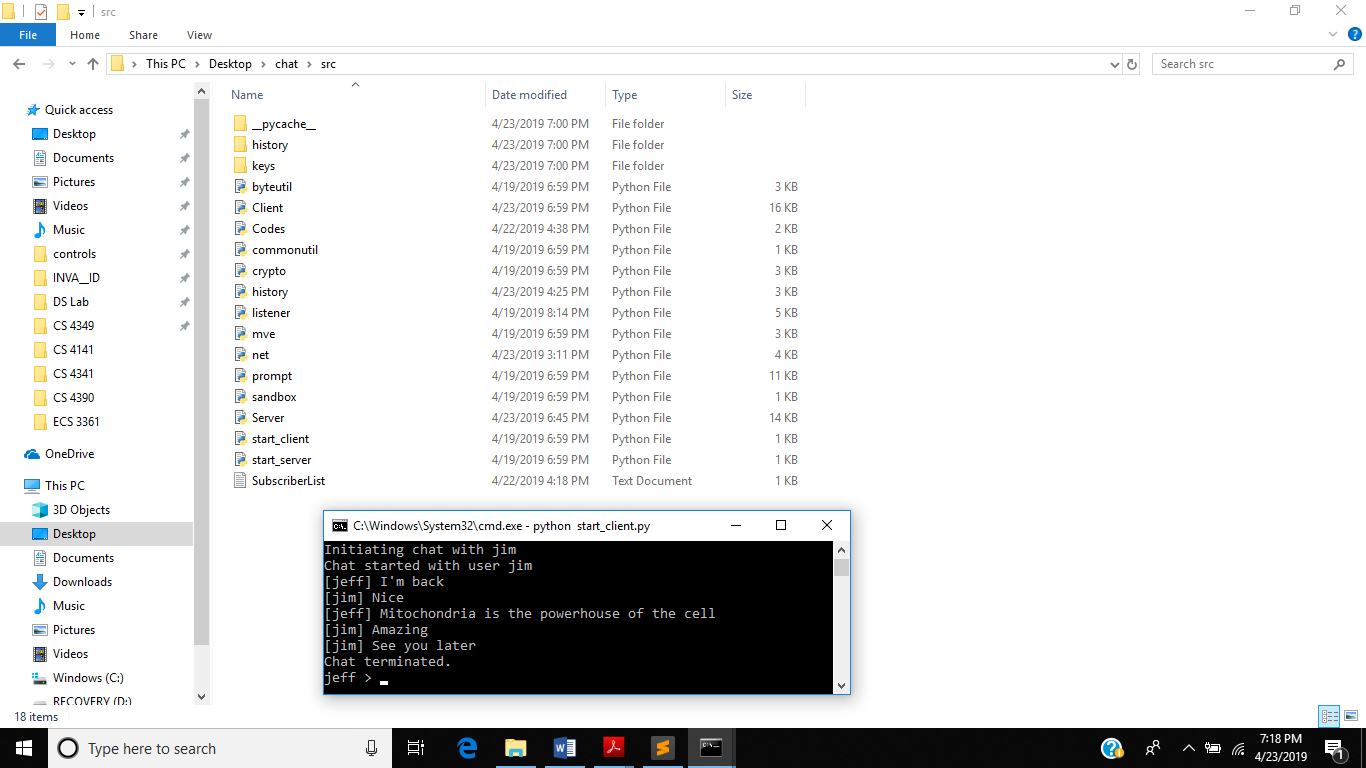
 Client-B Output:

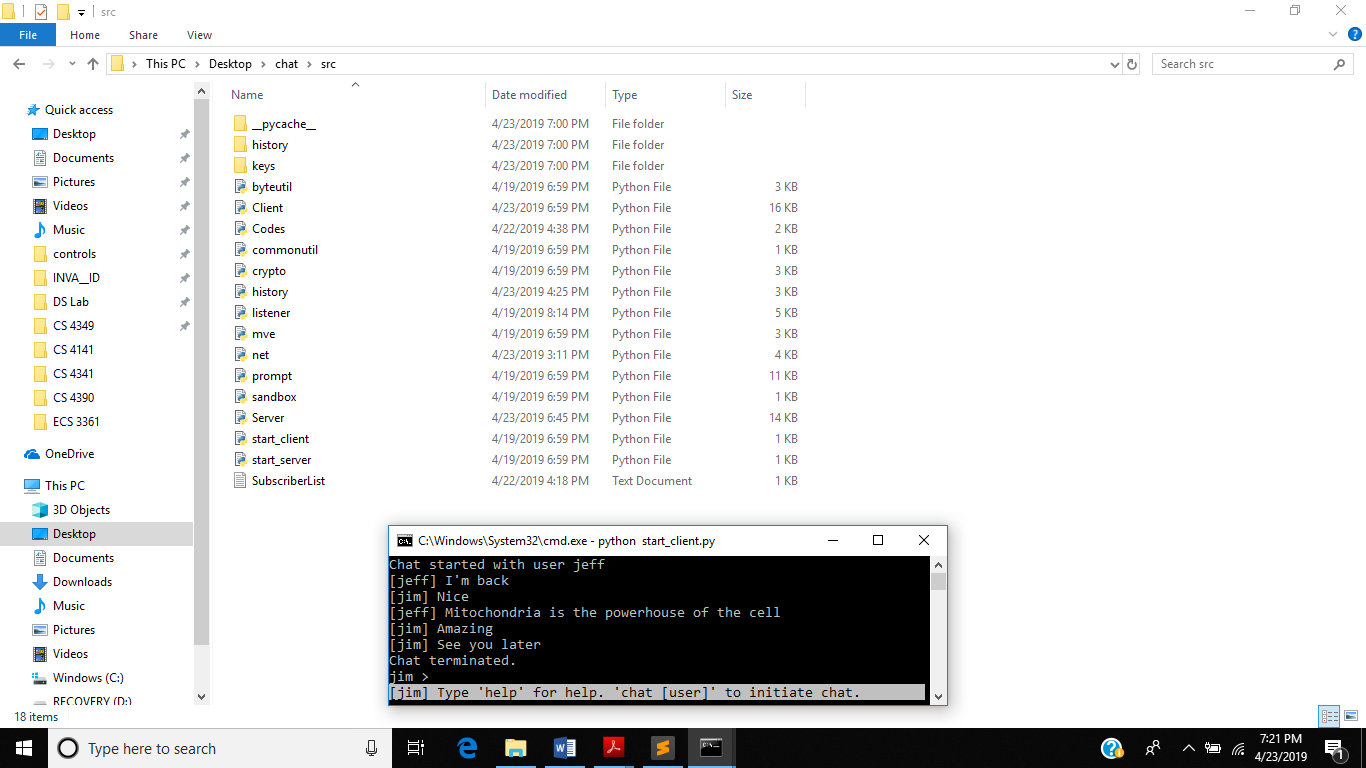
**Scenario 3b: Chat ended by receiving Client B**

Client-B Input:

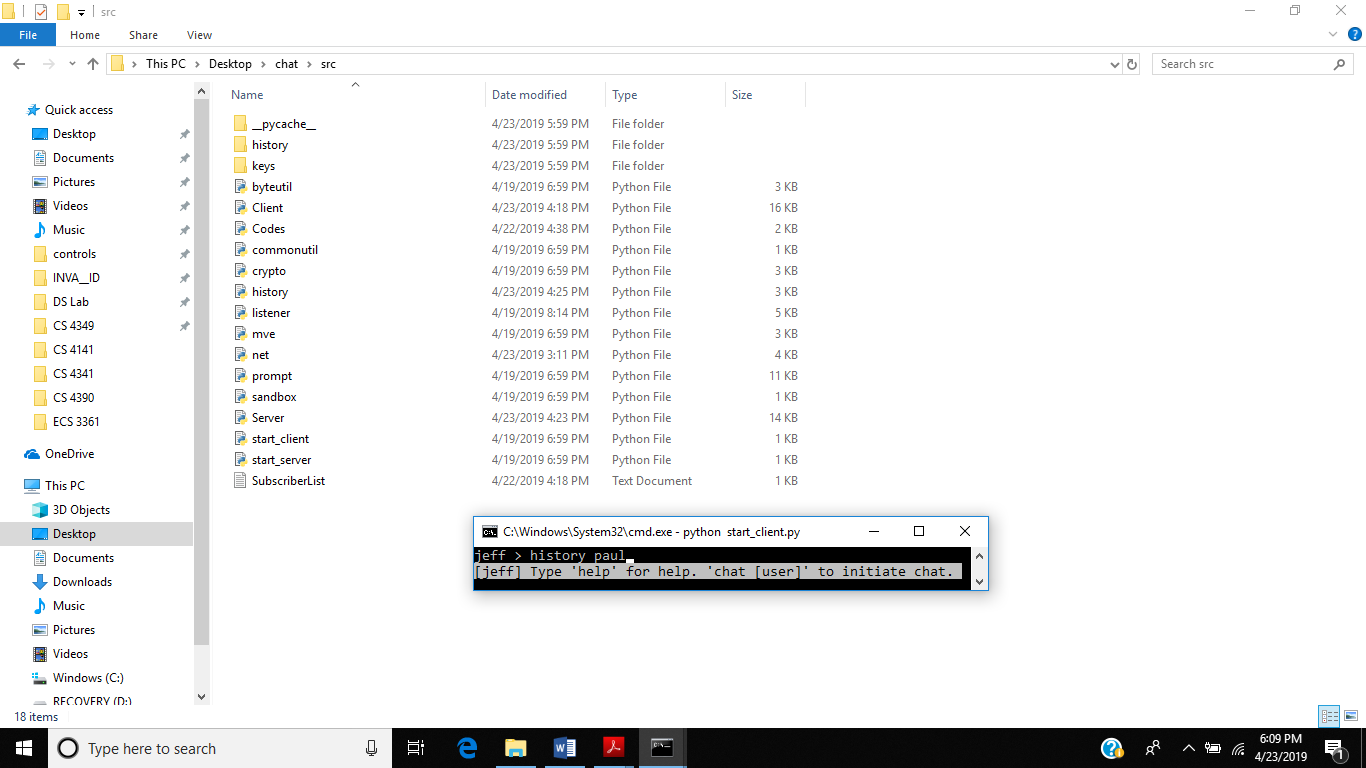


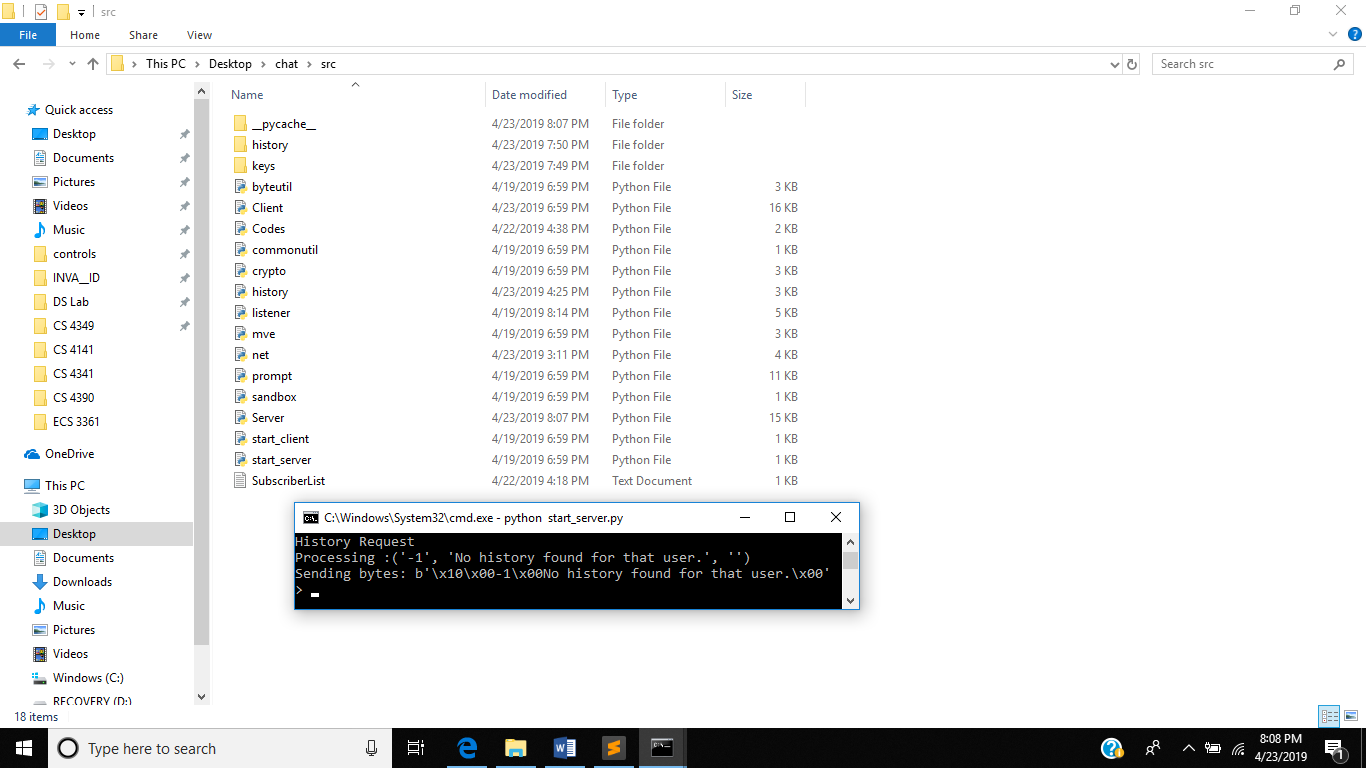
Server:

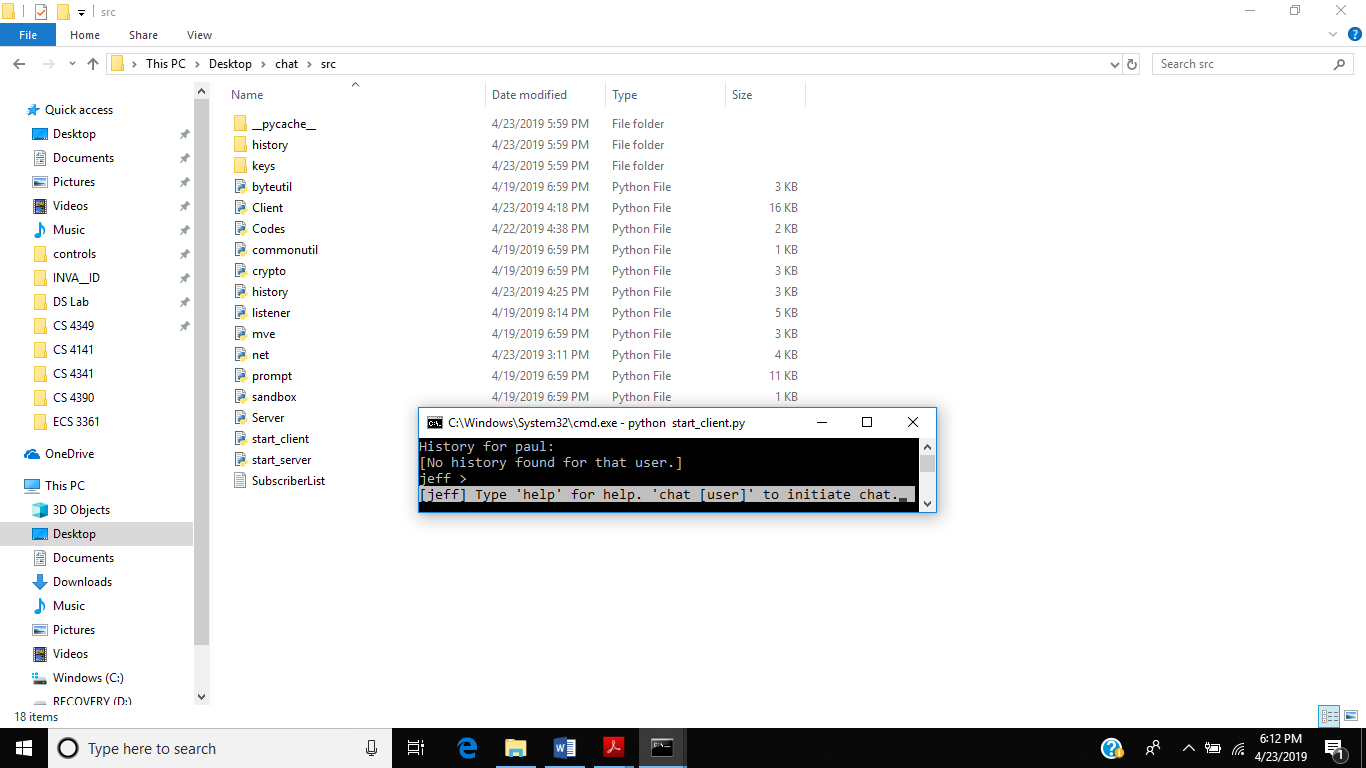
 Client-A Output:

 Client-B Output:

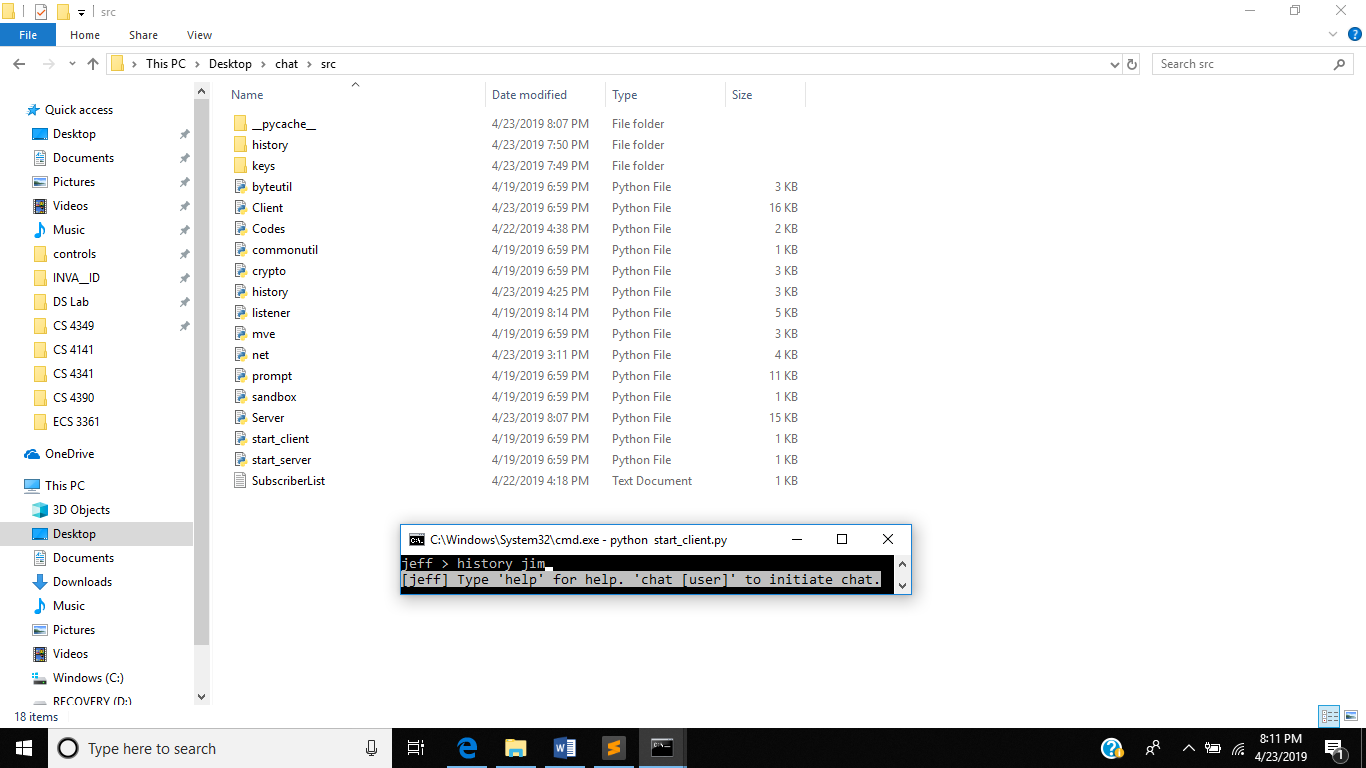
**Scenario 4a: Client A attempts to retrieve nonexistent history with Client B**

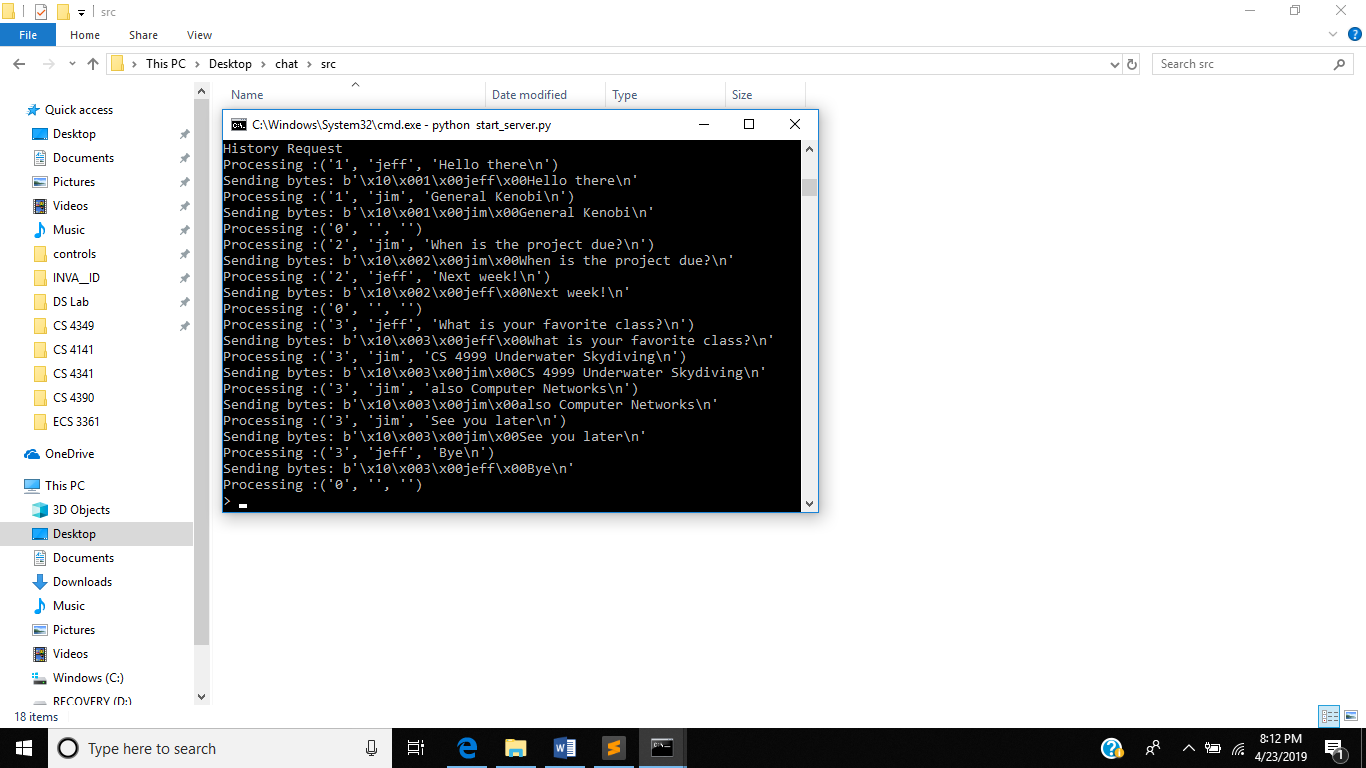
 Client Input:

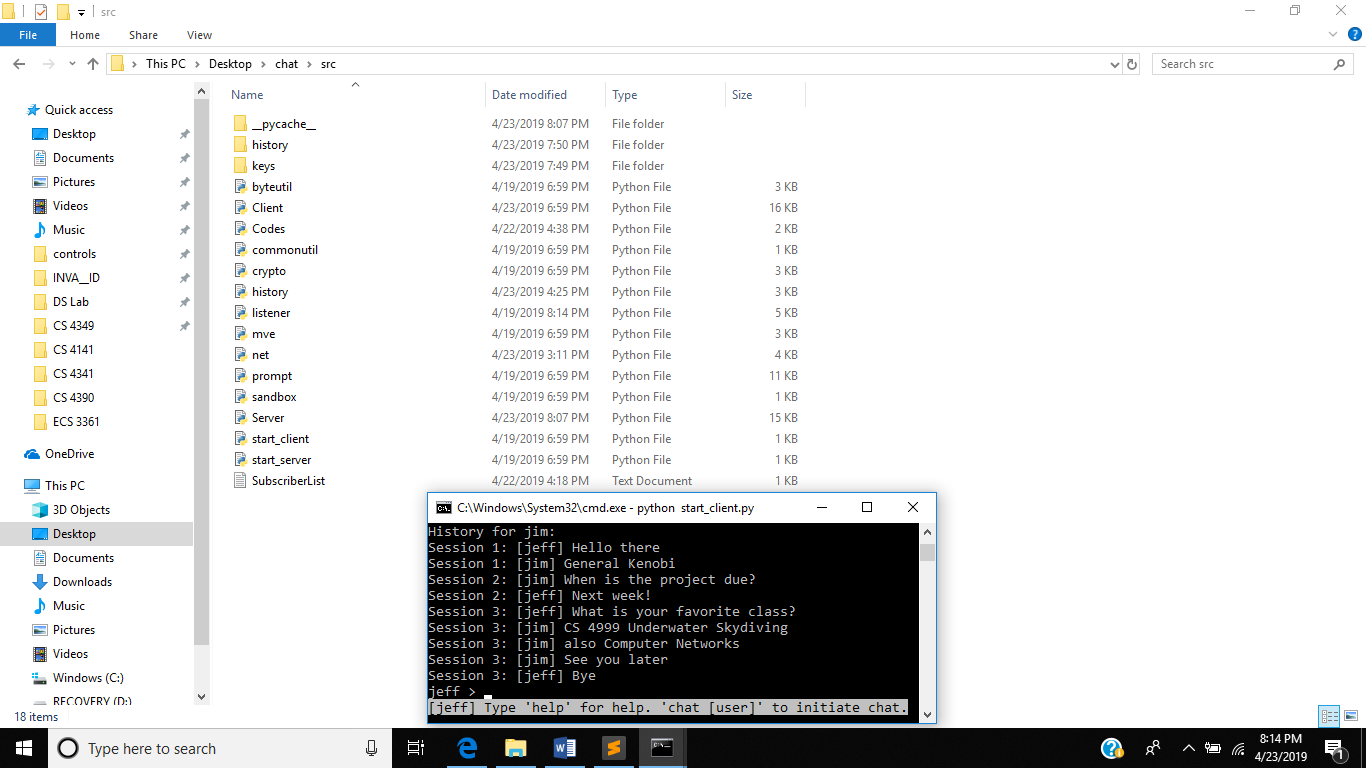
 Server:

Client Output:

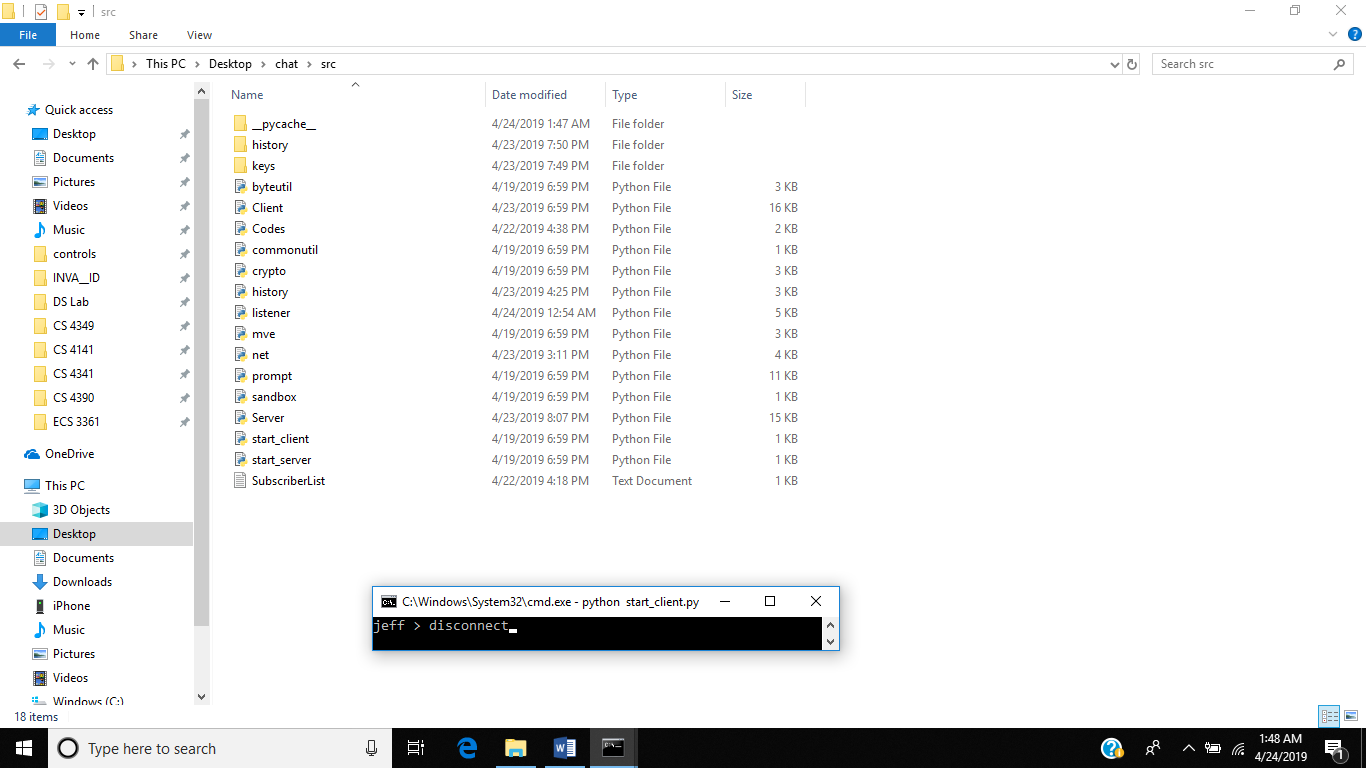
**Scenario 4b: Client A attempts to retrieve valid history with Client B**

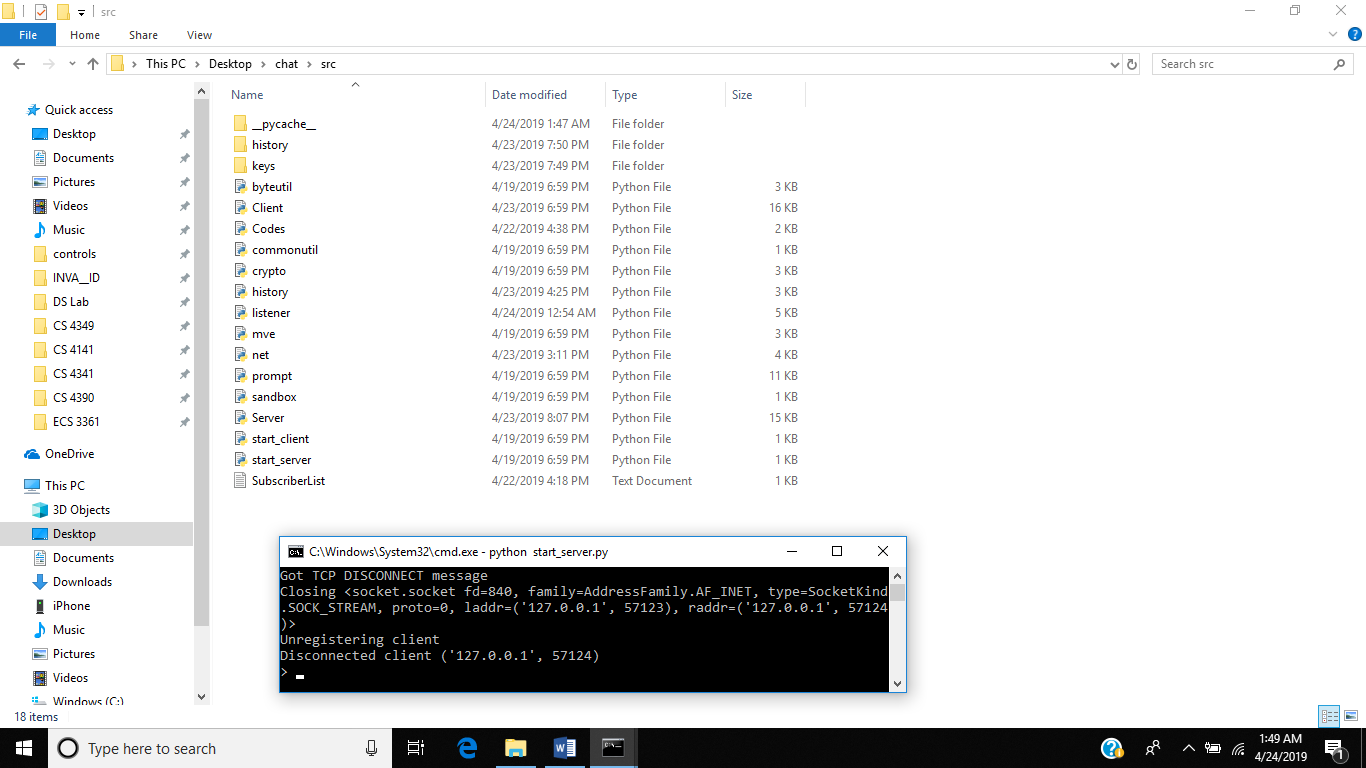
 Client Input:

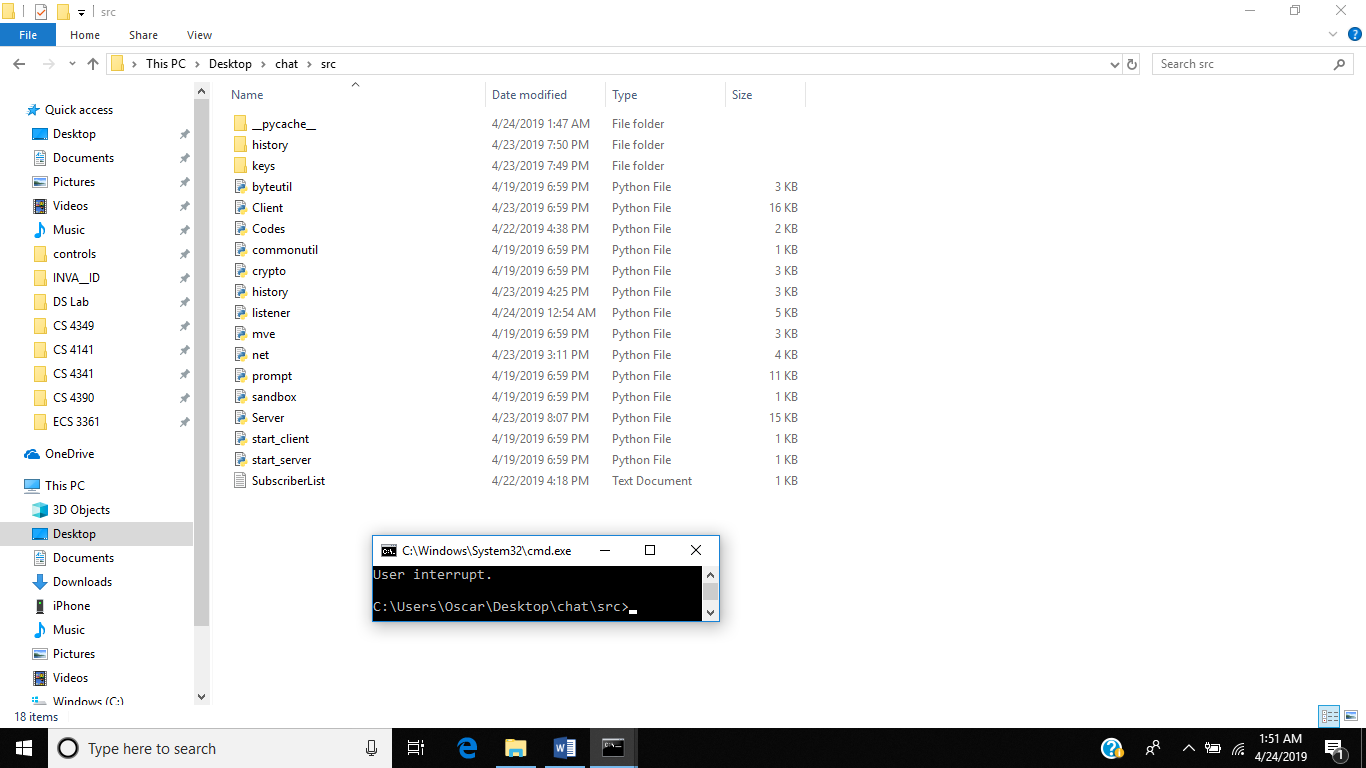
 Server:

Client Output:

**Scenario 5: Client A disconnects from the server**

 Client Input:

 Server:

 Client Output:

Challenges and Conclusion:

One of the most significant issues we faced in the project was finding a way to separate history response messages on the client side, as TCP is a byte-stream oriented protocol. When large messages were sent over TCP, they would essentially be massed together on the client end leading to garbled output and internal errors related to unexpected argument lengths. To solve this and keep track of the separate history response messages on the client end, we added our own message separator byte as a delimiter and added a “spin” mechanism to process an incoming byte stream. This allows the TCP listener on a client to correctly parse the bytes it receives and display the chat history as it was sent from the server.

Another byte-related issue we faced had to do with our implementation of the codes for each message. Each of the message types was assigned a byte code (e.g. b`\x01` for the HELLO message). For a time, we had the same byte for our message delimiter and RESPONSE messages for client authentication, leading to an ambiguous error stemming from how the bytes were parsed. We solved this by ensuring that each of these important codes or values had a unique byte to identify it, preventing such issues related to ambiguity.

Finally, a small but interesting issue we found was related to our implementation of TCP listeners on the server using threads. These would essentially run a routine in an infinite loop, waiting for incoming bytes from a client TCP socket. When disconnecting a client, we would find that an error was being thrown on the server. However, no functionality would actually be disrupted. The underlying reason was that when the client was no longer connected, the TCP listener would attempt to listen on a now-closed socket. This was fixed by catching the error and properly terminating the listener thread.

This project vastly improved our knowledge regarding the UDP and TCP protocols and taught us how to properly use them in an application. It also allowed us to improve our project management skills since we synchronized our activities in the project using GitHub for version control. Finally, working on this project gave us great insight on network programming and client-server architectures in particular, which are instrumental parts of modern computing.